Framework Migration of a Cloud-based Web App Backend

MASTER THESIS

Marco Martin Härtl

Submitted on 30 September 2024

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

Supervisor:
Dr. Andreas Kaufmann
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 30 September 2024

License

Erlangen, 30 September 2024

Abstract

Software undergoes an evolutionary process in which it adapts to changing requirements and opportunities. This thesis exemplifies and actively shapes the evolution of QDAcity, a cloud-based web application for qualitative data analysis in research. One dimension of this evolution is the technology stack, which frequently changes due to the rapid advancement in software development. New or changing requirements, along with inadequate or missing maintenance of dependencies, increase pressure on development teams. After about 10 years, it became necessary to replace the existing backend framework of QDAcity because support for the framework and the underlying cloud infrastructure was discontinued. Thus, this thesis describes the migration of QDAcity's backend framework to a more modern and long-term supported solution, which also lays the foundation for restructuring the cloud infrastructure. The focus is particularly on cost-sensitive preservation of all functionality of the web application and ensuring non-functional requirements. The transition to a new backend framework varies with each product and, therefore, requires a migration plan tailored to QDAcity, which is presented here. Additionally, the outcomes of the software architecture and technical design are presented and evaluated. Furthermore, the opportunities for further modernization of the technology stack arising from the migration are discussed.

Contents

List of Figures

List of Tables

Acronyms

API	application programming interface
Auth	authentication and authorization (no exact definition)
CaaS	containers as a service
CD	continuous delivery
CES	Collaborative Editing Service
CI	continuous integration
CPU	central processing unit
CRUD	create, read, update, delete
CSRF	cross-site request forgery
DAO	data access object
DB	database
DevOps	development and operations (no exact definition)
DI	dependency injection
FRQ	functional requirement
GAPI	Google API
GCS	Google Cloud Storage
Gen	generation
HTML	hypertext markup language
HTTP(S)	hypertext transfer protocol (secure)
IaaS	infrastructure as a service
ID	identity

IoC	inversion of control
ISO	International Organization for Standardization
JAR	Java archive
JSON	JavaScript object notation
JSP	Jakarta server pages
JWT	JSON web token
JVM	Java virtual machine
LTS	long-term support
LDAP	lightweight directory access protocol
MiB	mebibyte
MVC	model view controller
NPM	Node package manager
OS	operating system
PaaS	platform as a service
PoC	proof of concept
POJO	plain old Java object
POM	project object model
QDA	qualitative data analysis
QRQ	quality requirement
REST	representational state transfer
SaaS	software as a service
TLS	transport layer security
TRQ	technical requirement
UI	user interface
UML	unified modeling language
URL	uniform resource locator
WAR	web application archive
XML	extensible markup language
YAML	yet another markup language (no exact definition)

1 Introduction

Software constantly evolves to meet changing needs and conditions. As new technologies emerge and requirements shift, software must adapt to stay relevant, functional, maintainable, and secure. This ongoing process of software evolution (Lehman, 1980) involves updating frameworks, tools, and infrastructure to ensure that systems remain efficient, secure, and scalable in a rapidly changing environment. This thesis explores the challenges and strategies involved in migrating an outdated backend framework to a modern solution, highlighting the importance of maintaining core functionality while adapting to new technical demands. The product QDAcity was chosen as an example of such a framework migration.

1.1 Product QDAcity

QDAcity[1] [Q'dacity] is a powerful cloud-based web application tailored for qualitative data analysis[2] (QDA), offering robust features such as interview analysis and transcription via integrated speech-to-text software. Researchers can upload relevant files, leverage analytic statistics, and benefit from automated evaluation of the projects' maturity to streamline their analysis. Designed for both individual and collaborative group work, QDAcity allows users to efficiently manage projects together, making it ideal for research teams working on shared data. The tool also supports educational purposes, allowing instructors to organize courses and offer practical exercises. With secured storage and backup, the platform ensures data integrity while its cross-platform support allows for seamless access without the need for installation.

1.2 Thesis Goals

The main goal of this thesis is the migration of the QDAcity backend from an outdated and no longer supported framework to a modern, powerful, and long-term

[1]https://qdacity.com
[2]https://qdacity.com/qualitative-data-analysis/

supported framework. Moreover, this includes further technology changes in the backend and frontend components and major system design refactorings. Unlike the design, the QDAcity architecture, the system behavior, the application programming interface (API), the integration, the deployment, and the surrounding infrastructure should mostly remain unchanged. A very important aspect of this work is the solution strategy with its migration plan, bridging the gap between the legacy and the migrated state.

1.3 Thesis Type

The focus of this master thesis is on performing and presenting traditional engineering work, which involves the design and implementation of software within QDAcity as an example application. This so-called engineering thesis[3] requires the application of core engineering principles and technical expertise to develop functional solutions. Through careful research, development, and analysis, the thesis showcases the students' ability to bridge theory with practical implementation. This type of work not only highlights their skills in software engineering but also their contribution to solving real-world engineering challenges.

1.4 Thesis Structure

The structure of the thesis starts with this introduction and proceeds with the functional and non-functional requirements of the framework migration in chapter 2. Chapter 3 sketches the architecture of QDAcity, while chapter 4 gives insights into the system design and the implementations, performed in the practical part of this thesis. Finally, chapter 5 and chapter 6 present a critical review of the solutions and tasks that remain, before drawing a final conclusion in chapter 7.

[3]https://oss.cs.fau.de/theses/structure-content/

2 Requirements

This chapter defines the functional (see chapter 2.1) and non-functional requirements (see chapter 2.2) of the backend framework migration. The structure and wording of the requirements follow the MASTeR brochure (SOPHISTen, 2024), defining templates for requirements engineering. The following requirements present guidelines for the practical part of the thesis and objectives for the evaluation in chapter 5.

2.1 Functional Requirements

This chapter describes the functional requirements of the migrated API and backend configuration. Therefore, each requirement aligns with the FunctionalMASTeR template (SOPHISTen, 2024), which specifies a simple and precise sentence order (see figure 2.1). The sentence starts with an optional condition in the template structure of figure 2.2, while the second part is the declaration of the system name, which should be unique. Next, a keyword is chosen from the set of "shall", "should" or "will", which defines the legal obligation of the sentence. Requirements with "shall" are musts, with "should" nice-to-have, and "will" is optional. Requirements with "shall" are mandatory and must be fully implemented. Requirements with "should", while not compulsory, represent wishes and improve satisfaction if implemented. Requirements with "will" capture intentions for future features and therefore are not necessary for this thesis. The following part decides whether the functionality is an interface requirement, an automatic system activity, or a user interaction based on the type of phrase that is used (i.e., the phrases in figure 2.1 from top to bottom). At the end, the sentence contains a process verb describing the function and an object.

The most important functional objective of a system migration is:

FRQ-1 As soon as the migration is completed, the new backend should provide the user with the ability to use the web application as before.

Thus, the user should not even recognize the changes under the surface except for some non-functional improvements.

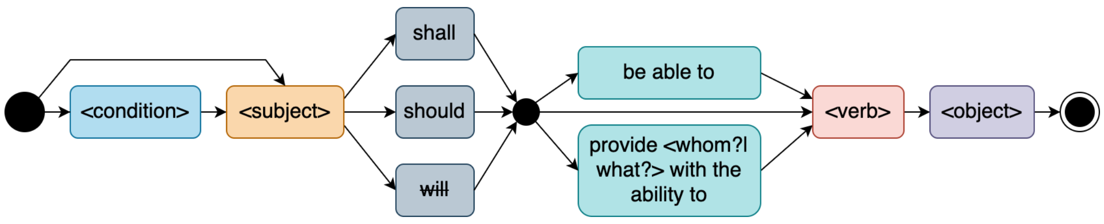


Figure 2.1: FunctionalMASTeR template for functional requirements (SOPHISTen, 2024)

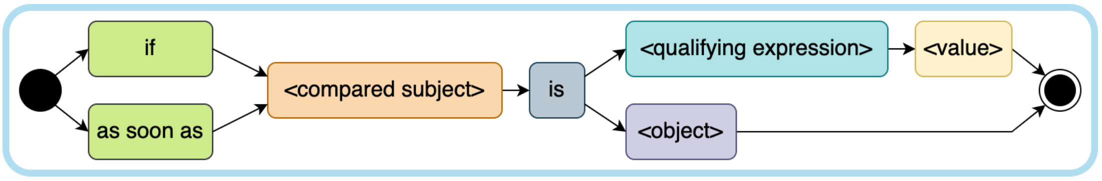


Figure 2.2: Custom template combining ConditionMASTeR, LogicMASTeR, and EventMASTeR for requirement conditions (SOPHISTen, 2024)

2.1.1 API Specification

The following list contains all functional requirements of the QDAcity API.

FRQ-2 Each endpoint shall define a unique URL path mapping.

FRQ-3 Each endpoint shall define a unique group tag.

FRQ-4 Each endpoint operation shall define a unique operation ID.

FRQ-5 Each endpoint operation should define a summary containing the endpoint tag and the endpoint operation ID.

FRQ-6 If authentication is required for an endpoint operation, the endpoint operation shall define bearer authentication as a security requirement.

FRQ-7 If authentication is not required for an endpoint operation, the endpoint operation should define no authentication as a security requirement.

FRQ-8 Each endpoint operation shall define only one HTTP method.

FRQ-9 Each endpoint operation shall define a unique URL path mapping.

FRQ-10 Each endpoint operation shall accept only the defined path parameters, query parameters, headers, and the body from an HTTP request.

FRQ-11 If the result of an HTTP request is successful and has a valid content, the endpoint operation shall return the HTTP response code 200 (OK) and the HTTP response body.

FRQ-12 If the result of an HTTP request is successful and has no content, the endpoint operation shall return the HTTP response code 204 (No content) and no HTTP response body.

FRQ-13 If a parameter or the body of an HTTP request is invalid, the endpoint operation shall return the HTTP response code 400 (Bad request) and an error message in the HTTP response body.

FRQ-14 If the authentication result of an HTTP request is unauthenticated, the endpoint operation shall return the HTTP response code 401 (Unauthorized) and an error message in the HTTP response body.

FRQ-15 If the authorization result of an HTTP request is unauthorized, the endpoint operation shall return the HTTP response code 401 (Unauthorized) and an error message in the HTTP response body.

FRQ-16 If the mapping of an HTTP request to an endpoint operation is not possible, the web server shall return the HTTP response code 404 (Not found) and an error message in the HTTP response body.

FRQ-17 If the handling of an HTTP request is interrupted by an error, the endpoint operation shall return by default the HTTP response code 500 (Internal server error) and an error message in the HTTP response body.

FRQ-18 If the result of an HTTP request is failed and has an error message, the endpoint operation shall return this message in the HTTP response body.

2.1.2 API Security

The following list contains all functional requirements concerning the security features of the QDAcity API.

FRQ-19 The backend shall be able to validate a JSON web token (JWT) sent as a bearer token in the authorization header of an HTTP request.

FRQ-20 The backend shall be able to deliver a registered QDAcity ID token for a login request with a valid Google ID token of a registered user account.

FRQ-21 The backend shall be able to deliver a registered QDAcity ID token for a login request with a valid email and password of a registered user account.

FRQ-22 The backend shall be able to deliver an anonymous QDAcity ID token for a login request without any credentials.

FRQ-23 If access to an endpoint operation is allowed for Google ID tokens, the backend shall be able to permit an HTTP request with a valid Google

ID token of a registered user account.

FRQ-24 If access to an endpoint operation is allowed for registered QDAcity ID tokens, the backend shall be able to permit an HTTP request with a valid registered QDAcity ID token.

FRQ-25 If access to an endpoint operation is allowed for anonymous QDAcity ID tokens, the backend shall be able to permit an HTTP request with a valid, anonymous QDAcity ID token.

FRQ-26 If access to an endpoint operation is allowed without any tokens, the backend shall be able to permit an HTTP request without any credentials or tokens.

2.1.3 API Client for Manual Testing

The following list contains all functional requirements for an API client, which allows the QDAcity team to test the QDAcity API manually.

FRQ-27 The API client for manual testing shall provide the developer with the ability to prepare and send HTTP requests to the backend.

FRQ-28 The API client for manual testing shall provide the developer with the ability to receive and inspect HTTP responses from the backend.

FRQ-29 The API client for manual testing should provide the developer with the ability to use predefined structures of the HTTP request body.

FRQ-30 The API client for manual testing should provide the developer with the ability to inspect the data models.

FRQ-31 The API client for manual testing should provide the developer with the ability to acquire information about the endpoint operations.

FRQ-32 The API client for manual testing shall provide the developer with the ability to authenticate and authorize HTTP requests.

FRQ-33 The API client for manual testing should provide the developer with the ability to find endpoint operations via their operation ID.

FRQ-34 The API client for manual testing should provide the developer with the ability to choose and display a specific endpoint.

2.2 Non-Functional Requirements

After covering the functional requirements, this chapter presents the non-functional requirements. These are divided into technological (see chapter 2.2.1) and quality (see chapter 2.2.2) requirements. This distinction originates from the MASTeR brochure (SOPHISTen, 2024), which also defines some more categories, not covered in this thesis. The technological and quality requirements use the so-called PropertyMASTeR (see figure 2.3). Similarly to the functional requirements, this template uses standardized keywords and phrases reducing the uncertainty of misinterpretation and simultaneously sharpening the definition of requirements.

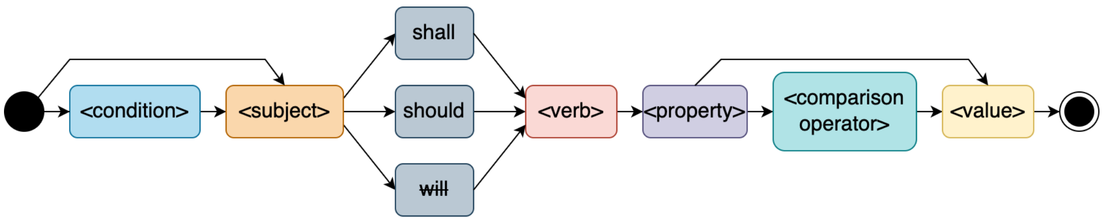


Figure 2.3: PropertyMASTeR template for technological and quality requirements (SOPHISTen, 2024)

The PropertyMASTeR (SOPHISTen, 2024) starts with an optional condition in the template structure of figure 2.2. The second building block defines the subject of matter. Next, the legal obligation is specified by a keyword from the set "shall", "should" or "will" and combined with a verb. At the end of the sentence, there remains a property-value pair with a potentially optional qualifying expression in between those two parts.

2.2.1 Technological Requirements

The technological requirements describe rules for the environment, quantity structure, interfaces, and components. In this context, the thesis uses only the environmental and interface-level requirements.

Environment

TRQ-1 The backend shall use the host platform Google App Engine Gen 2 with bundled Gen 1 services.

TRQ-2 The backend shall use the programming language Java version 8 for compilation and version 17 at runtime for the Java virtual machine (JVM).

TRQ-3 The backend shall use the package manager Maven of major version 3.

Interfaces

TRQ-4 The API of the backend shall be compliant with the API specification standard OpenAPI version 3.0.

TRQ-5 The access control of the backend shall use the token format JWT.

2.2.2 Quality Requirements

The ISO 25010:2011[1] standard distinguishes between the product quality model and quality in use. The latter interprets quality from the users' point of view, which is less applicable to APIs because the user certainly does not directly interact with it. Apart from that, the product quality model describes quality as a measurable characteristic of the software product. Therefore, the quality dimensions of the ISO standard contain seven "true" non-functional categories performance, compatibility, usability, reliability, security, maintainability, and portability. The missing one would be functional suitability, which can be seen as a part of functional requirements as it enforces the software to satisfy the real functional needs of the user.

Performance

QRQ-1 The API client in the frontend shall cache the OpenAPI specification until the API version changes.

QRQ-2 The web application hosted on App Engine shall not reach a CPU utilization of more than 20%.

QRQ-3 The web application hosted on App Engine shall not reach a memory usage of more than 600 mebibytes (MiB).

QRQ-4 The web application hosted on App Engine shall not reach a response latency of more than 1s, at maximum 2s.

Compatibility

QRQ-5 Each endpoint operation should define its operation ID equal to the operation name of the corresponding legacy endpoint operation.

QRQ-6 Each endpoint operation should define its URL path mapping equal to the URL path mapping of the corresponding legacy endpoint operation.

QRQ-7 Each endpoint operation should define its HTTP method equal to the HTTP method of the corresponding legacy endpoint operation.

[1]https://www.iso.org/standard/35733.html

QRQ-8 Each endpoint operation should define its HTTP request parameters equal to the HTTP request parameters of the corresponding legacy endpoint operation.

QRQ-9 Each endpoint operation should define its HTTP request body equal to the HTTP request body of the corresponding legacy endpoint operation.

QRQ-10 Each endpoint operation should define its HTTP response codes equal to the HTTP response codes of the corresponding legacy endpoint operation.

QRQ-11 Each endpoint operation should define its HTTP response body equal to the HTTP response body of the corresponding legacy endpoint operation.

QRQ-12 Each endpoint operation should define its required permissions equal to the required permissions of the corresponding legacy endpoint operation.

Usability

QRQ-13 The API client for manual testing should provide local storage for the ID token that is persistent across sub-pages and page reloads.

Reliability

QRQ-14 The web application hosted on App Engine shall allow a rollback to an older app version as before with the legacy web application.

QRQ-15 The API of the backend should have an API version that is independent of the app version.

Security

QRQ-16 The HTTP communication between API clients and the QDAcity API shall ensure encryption via transport layer security (TLS).

QRQ-17 If data sent to an endpoint operation is confident, the endpoint operation shall accept private data only via its HTTP request body.

QRQ-18 If an endpoint operation is permission-dependent, the endpoint operation shall have permission control via the authorization mechanism.

QRQ-19 If the application host environment is a production App Engine instance, the web application should set the lowest logging level to "info".

QRQ-20 If the application host environment is a local devserver, a test App Engine instance, or a pipeline job container, the web application should set the lowest logging level to "debug".

QRQ-21 The backend should ensure that the number of vulnerable dependencies is equal to zero.

Maintainability

QRQ-22 The backend shall have an integration and deployment process that is mostly automated and continuous via GitLab CI.

QRQ-23 The backend should reach a branch code coverage via tests of at least 60%.

QRQ-24 The migration documentation should explain development-related details in the GitLab Wiki.

QRQ-25 The migration documentation should explain project- or product-related details in the GitLab Wiki.

QRQ-26 The migration documentation should explain each endpoint operation in the API client for manual testing.

QRQ-27 The migration documentation should explain the migration details in this thesis.

Portability

QRQ-28 The web application hosted on App Engine shall possess an app setup that is adaptable to native Google App Engine Gen 2 without bundled Gen 1 services.

QRQ-29 The web application hosted on App Engine shall possess an app integration that is adaptable to native Google App Engine Gen 2 without bundled Gen 1 services.

QRQ-30 The web application hosted on App Engine shall possess an app deployment that is adaptable to native Google App Engine Gen 2 without bundled Gen 1 services.

3 Architecture

This chapter delineates the software architecture of QDAcity and provides a comprehensive overview of the fundamental system design, used technologies, patterns, styles, and architectural decisions, that underpin the implementation. It follows the overall structure of the arc42[1] template and thus encompasses the solution strategy (see chapter 3.1) for the backend migration to a new technology stack, the backend (see chapter 3.2) and frontend (see chapter 3.3) architecture on a component level, the integration and deployment (see chapter 3.4) of the project artifacts, and the architecture decisions (see chapters 3.5 and 3.6). It offers a cohesive view of how these elements interact to achieve the intended system objectives. Moreover, it establishes a clear understanding of the used components and technologies, offering abstract and more detailed perspectives, which lay a solid foundation for the subsequent chapter 4 about design and implementation.

3.1 Solution Strategy

The solution strategy outlines the general approach and provides a high-level summary of the migration strategy, employed to address the primary requirements, both functional and non-functional. It serves as a guiding framework, which influences the system's structure and behavior, ensuring that the architecture effectively meets its intended goals. This chapter starts with a short overview of the QDAcity system (see chapter 3.1.1), followed by a comparison of the legacy and migrated technology stacks (see chapters 3.1.2 and 3.1.3). Lastly, the chosen migration plan (see chapter 3.1.4) and the incremental development approach (see chapter 3.1.5) cover the methodology of the migration process.

3.1.1 System QDAcity

The system QDAcity consists of three main components, a frontend, a backend, and an additional service called Collaborative Editing Service (CES), as illustrated in figure 3.1. This component diagram and most following diagrams based

[1] https://arc42.org/overview

on the unified modeling language[2] (UML) distinguish four primary colors, blue for the frontend, red for the CES, green for the added or migrated backend parts, and orange for the legacy core of the backend. Beginning with the frontend, it is a client-side web application that handles the user inputs and visualizes information via rendering an HTML document in the browser. The backend is a server-side web service, which holds the most business logic and accesses the data, stored in a database system (i.e., Google Datastore[3]). In between those two modules, there is the CES. For some requests, it is a proxy between the frontend and the backend, for others, it is a web service hosting the CES API. The API can be accessed by the backend or frontend. The CES is used for retrieving and storing data in Google Cloud Storage[4] (GCS), concerning the collaborative work of multiple users. Lastly, the frontend delivers a QDAcity user interface (UI) to the user via a web client (i.e., the web browser).

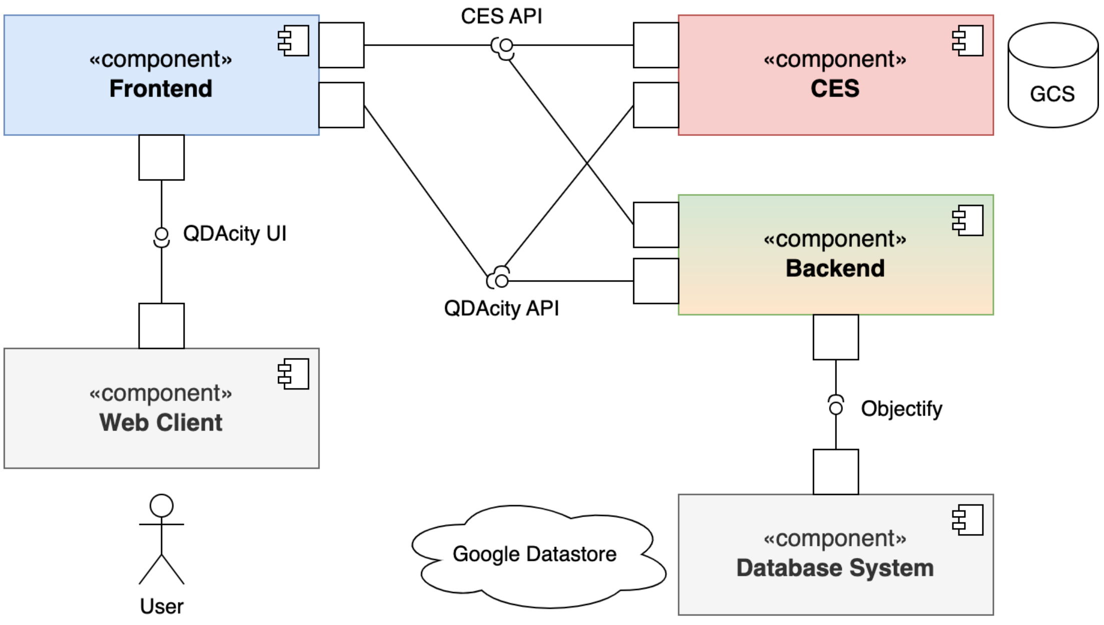


Figure 3.1: UML component diagram of the QDAcity system

3.1.2 Legacy Technology Stack

The legacy QDAcity backend and frontend were hosted in a standard Java runtime environment[5] of Google App Engine Gen 2 with bundled Gen 1 services, which are already deprecated. Hosting an application in a foreign cloud infrastructure like Google Cloud gains some provider-dependent services such as

[2]https://www.omg.org/spec/UML/
[3]https://cloud.google.com/datastore/docs/concepts/overview
[4]https://cloud.google.com/storage/docs
[5]https://cloud.google.com/appengine/docs/standard/java-gen2/runtime

Google Datastore and Google Speech-to-Text[6]. The latter is a service converting speech recordings to text transcripts. In addition, Google App Engine provides a Jetty[7] 9 web server, which runs the servlets of QDAcity.

The legacy backend was based on the programming language Java 8 for compilation, the JVM version 17 at runtime, and the Google Endpoints[8] framework for defining endpoints.

The legacy frontend was based on the programming language JavaScript ES6 and the React[9] framework. Besides other dependencies, it contains Google API (GAPI) Client[10] for requesting the QDAcity API and Google APIs.

3.1.3 Migrated Technology Stack

The migrated technology stack (see table 3.1) of the QDAcity backend and frontend still contains the Google App Engine Gen 2 with bundled Gen 1 services because the legacy backend framework is not compatible with the native Google App Engine Gen 2, but was necessary for the migration process (see chapter 3.1.4). Moreover, every used Google service and the Jetty 9 web server remain the same.

The migrated backend is also based on Java 8 (and JVM 17), but on the Spring Boot[11] 2 framework. Due to the migration process, it was necessary to use the older major version 2 of Spring Boot because it no longer supports Java 8 since major version 3, which is a prerequisite to the legacy backend code. These outdated versions should be upgraded in a future migration task following the Spring Boot documentation. The Spring Boot framework encapsulates some modules such as Spring Boot Starter Web, Security, Jetty, and Test for future Spring-based test suites (see chapter 6.2.1). Some of them contain vulnerable sub-dependencies, which were explicitly analyzed and fixed. Another newly introduced dependency is SpringDoc-OpenAPI[12], which combines OpenAPI, SwaggerUI, and other Swagger dependencies for specifying and documenting the QDAcity API. The added Lombok[13] dependency compiles its annotations to Java code and thus streamlines the application code.

The migrated frontend is still based on JavaScript ES6 and React but contains a new API client called Swagger Client[14] (also known as SwaggerJS) to access

[6]https://cloud.google.com/speech-to-text
[7]https://jetty.org
[8]https://cloud.google.com/endpoints/docs/frameworks/java
[9]https://react.dev
[10]https://github.com/google/google-api-javascript-client
[11]https://docs.spring.io/spring-boot/index.html
[12]https://springdoc.org/v1/
[13]https://projectlombok.org
[14]https://www.npmjs.com/package/swagger-client

the Spring Boot backend via the QDAcity API. This was essential because the backend is no longer based on the Google Endpoints framework and therefore not accessible via GAPI Client, which handles only APIs recognized by Google.

Technology	Dependency	Version
Spring Boot	spring-boot-dependencies	2.7.18
SpringDoc-OpenAPI	springdoc-openapi-ui	1.8.0
Lombok	lombok	1.18.32
Swagger Client	swagger-client	3.28.2

Table 3.1: Technology stack of new dependencies with the introduced versions

3.1.4 Migration Plan

The migration plan shown below describes the steps that were needed to modernize the backend framework.

1. Integration of the new backend framework
   (a) Preparation of the backend project
   (b) Adaptation of the authentication and authorization mechanism
   (c) Integration of new API client for development
   (d) Customization of the exception handling and data serialization
   (e) Duplication of the endpoint classes (facade pattern)
   (f) Completion of the web server configuration (including frontend hosting)
2. Switch of the frontend to the new API client
   (a) Preparation of the frontend project
   (b) Addition of a new central API client
   (c) Migration of the endpoint services and hard-coded fetches
   (d) Adjustment of the token handling to the new client
   (e) Integration of response interceptors to adapt the responses
3. Extraction of the legacy backend framework

Step one integrates the Spring Boot framework in parallel to the remaining legacy Google Endpoints framework. Therefore, it prepares the backend project initially and incorporates increasingly more modules of Spring Boot via an incremental approach. This includes adapting the backend security as authentication and authorization, integrating a new API client for developers, customizing Spring Boot's exception handling, duplicating the endpoint classes, and completing the setup of the Jetty web server. The duplication of the endpoint classes follows a pattern called facade (Gamma et al., 1994), where a new endpoint layer is added in front of the existing API. This has the advantage that the legacy API remains unchanged and thus the Spring Boot backend can be hosted separately during the migration process. As another benefit, the Spring Boot endpoint classes must not copy the logic but can reuse the code of the legacy endpoints by simply calling the corresponding methods.

Step two integrates a new API client in the frontend called Swagger Client. It enables the frontend to request the new Spring Boot backend while preserving the ability to call the legacy backend via the GAPI client. At first, the frontend project is prepared by installing the new package and introducing a new central logic for using the client. After the initial setup, the incremental migration of the endpoint services and hard-coded fetches takes place. In the end, the token handling must be adjusted and the API responses need adapter solutions for compatibility with the legacy data formats and structures.

Step three extracts the legacy Google Endpoints framework and cleans the project of QDAcity. Most of the tasks are already done but are not explicitly described in this thesis because these are mindless work.

3.1.5 Incremental Development

The development process during the migration process follows an incremental approach. This means that all development tasks should be well structured into smaller work packages. The changes for these work packages are then reviewed, merged, deployed, and released regularly, i.e., weekly or at least monthly. Thus, it prevents a big bang release, where all migration work would culminate in a single delivery at the end of the process. This would come with the big risk of breaking the deadline or delaying the migration. The incremental development approach is therefore favored to minimize this risk.

3.2 Backend

This chapter presents a detailed examination of the QDAcity backend on an abstract level. The architecture is designed to facilitate efficient, scalable, and secure communication between client applications and the server, adhering to some principles of representational state transfer[15] (REST). By leveraging a stateless, modular approach, the backend architecture ensures robust performance and ease

[15]https://restfulapi.net

of maintenance. This chapter explores the core components and layers of the architecture (see chapter 3.2.1), including their interactions and responsibilities. Later on, it provides a detailed view of the technologies (see chapters 3.2.2 to 3.2.7) employed for the design and implementation part in chapter 4.1.

3.2.1 Layered Backend Architecture

The QDAcity backend builds on the well-known layered system architecture (Richards, 2015) and divides the application into three main tiers. First of all, the presentation layer contains the API and the endpoints. The following application layer encapsulates the business logic and the persistence layer communicates with the database system (in the fourth database layer), in this case, Google Datastore.

Figure 3.2 contains these layers represented by the endpoint layer, the business layer, and the data access layer components.

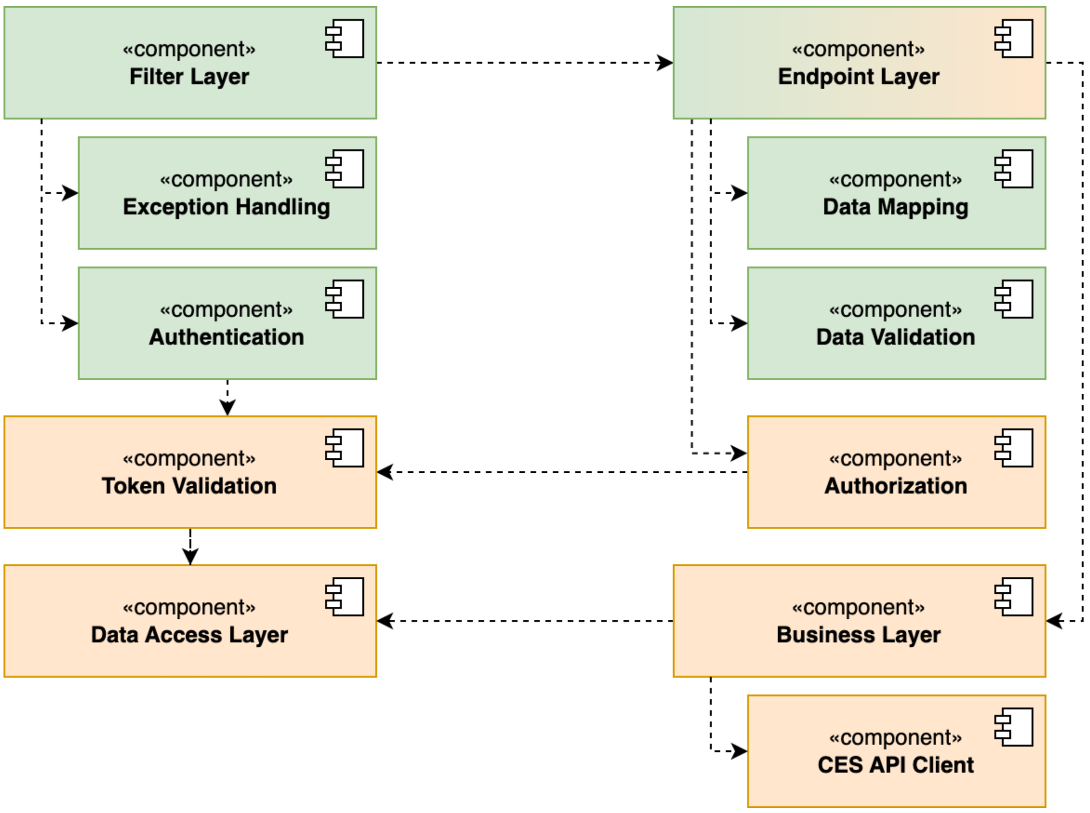

«component» Filter Layer
«component» Endpoint Layer
«component» Exception Handling
«component» Data Mapping
«component» Authentication
«component» Data Validation
«component» Token Validation
«component» Authorization
«component» Data Access Layer
«component» Business Layer
«component» CES API Client

Figure 3.2: Partial UML component diagram of the QDAcity backend

The architecture of the QDAcity backend comprises a mix of newly introduced components and legacy components with a clear focus on handling authentication, authorization, data access, and API communication. The green-colored components represent the new additions to the QDAcity API module, while the orange-colored ones signify the legacy components that continue to provide foundational services.

At the core of the backend is the filter layer, which serves as an entry point for requests. As requests pass through this layer, the system filters them based on predefined rules or conditions. The authentication component plays a critical role at this point, as it ensures that the incoming requests are authenticated by validating the identity of the user or system making the request. If an issue arises, the exception handling component is responsible for capturing and managing any errors or exceptions, preventing them from propagating further.

Once authentication is successful, the request is passed to the token validation component, which is a part of the legacy system. This component ensures that the request includes a valid token, verifying that the token is both legitimate and still in effect. After token validation, control is handed over to the data access layer, which facilitates access to the underlying data by ensuring that the backend can retrieve or modify information as required by the API calls.

On the side of request processing, the endpoint layer represents the interface between the backend and various services. It works closely with the data mapping and data validation components to ensure that the data in the requests is correctly interpreted and validated before being processed further. Data mapping is important for converting data between different formats, while validation ensures that the data conforms to the required standards and rules.

The authorization component, a legacy component, plays a critical role in ensuring that once the users are authenticated, they also have the correct permissions to access the requested resources or perform specific actions. This is distinct from authentication and involves verifying the users' access rights and roles. After this, the request is processed through the business layer, which holds the business logic of the application, ensuring that the operations performed adhere to the system's core functionality and rules.

Finally, the CES API client is responsible for communicating with the CES. It connects the backend to CES endpoints that the QDAcity backend might need to interact with, facilitating communication and data exchange between these systems.

In summary, the QDAcity backend architecture ensures that each request is thoroughly authenticated, authorized, and validated while handling data mapping and access with a clear separation between new and legacy components. This separation allows for a modular and maintainable system where new functionality can be added without disrupting the existing legacy infrastructure.

The QDAcity backend combines multiple levels of abstraction via the internal structure, the Spring Boot framework, and the Jetty web server. However, more details on the design and implementation of the QDAcity backend are covered in chapter 4.1.

3.2.2 Spring Framework and Spring Boot

The Spring[16] framework delivers a programming and configuration model for enterprise applications based on the Java programming language. It easily integrates with most of the surrounding technologies and the deployment environment, so that the developers can focus on the business logic of applications.

Spring Boot[17] and Spring are often used as synonyms but mean completely different things. While Spring is a generic framework for Java applications (as described before), Spring Boot is a wrapper around the Spring framework. It provides an opinionated configuration of Spring modules, which can be customized as needed. Furthermore, Spring Boot also imports and configures third-party dependencies, which it considers necessary or valuable. These dependencies and some Spring modules get combined into bigger Spring Boot Starter dependencies, which function as bundles for specific requirements, use cases, or even application types and are plug-and-play solutions. This "just run" fashion of an application framework is the core objective of Spring Boot.

Whereas, the Spring framework provides core technologies as dependency injection via the IoC container and Spring beans (see chapter 3.2.3). In addition, it is extended with valuable modules like Spring Web MVC (see chapter 3.2.4) that enable developing web applications as the QDAcity backend. Furthermore, Spring Security (see chapter 3.2.5) helps add authentication and authorization to the application. This variety of Spring modules and core technologies combined with the pre-configuration via Spring Boot is distinctive to most other Java frameworks, as detailed in chapter 3.5.1.

3.2.3 Spring Dependency Injection via IoC

Spring implements a dependency injection mechanism, where components define their dependencies via properties, constructors, or factory methods. The components do not instantiate these dependencies because the Spring IoC container[18] does it automatically during application start-up (see figure 3.3). The interface called ApplicationContext is a representation of the inversion of control (IoC) principle, which is the exact opposite of Spring beans[19]. It controls the instantiation of these beans and the location of their dependencies. Whereas, beans

[16]https://spring.io/projects/spring-framework
[17]https://spring.io/projects/spring-boot
[18]https://docs.spring.io/spring-framework/reference/core/beans/basics.html
[19]https://docs.spring.io/spring-framework/reference/core/beans/definition.html

directly construct instances (e.g., service locator pattern (Fowler, 2003)) and do not rely on the application context via IoC.

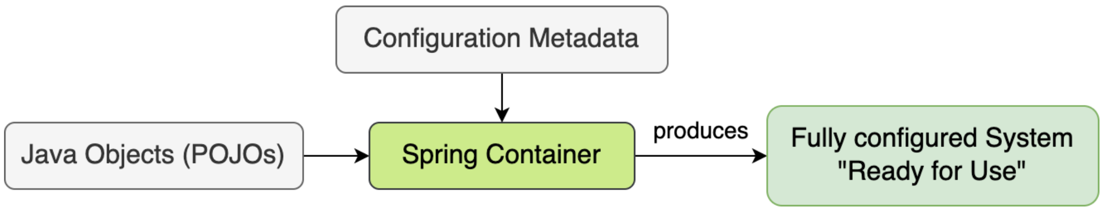


Figure 3.3: Spring IoC Container receiving plain old java objects (POJOs) and metadata to produce a fully configured system (Spring documentation)

In the Spring framework, these beans are configured Java-based via so-called configuration classes or annotation-based via configuration metadata on the component classes. Additionally, XML configurations (e.g., web.xml) or combinations of both are also possible but less common. The IoC container scans the application for bean metadata and thus creates the beans in the right order so that every bean gets its dependencies first. After this context preparation, all beans are available and the application is ready for use (see figure 3.3).

This dependency injection mechanism via IoC leverages the overhead for explicit instantiation of components (i.e., beans) from the bottom upwards. Furthermore, it helps decoupling bean creation from the actual business logic and reduces the risk of duplicate instances from the same class. For web applications such as the QDAcity backend, there also is a specialized variant of the ApplicationContext interface called WebApplicationContext.

3.2.4 Spring Web MVC

Spring Web MVC[20] is a submodule of the Spring framework for servlet-stack web applications and thus a web framework. The name contains the term model view controller (MVC), which describes a design pattern used to separate the model (i.e., data), the view (i.e., UI or API), and the controller (i.e., logic). (Fowler, 2003) Spring MVC is based on the servlet API and deployed to servlet containers as Jetty (or Tomcat). As many other frameworks, Spring MVC follows the front controller pattern (Fowler, 2003), where a central servlet called DispatcherServlet handles the request processing on a high level and delegates the actual work to the components. The DispatcherServlet requires a WebApplicationContext as its IoC container, which is automatically provided by Spring Boot (see figure 3.4).

Commonly, the whole Spring web application is structured into layers. These represent at first a representation or boundary layer called the controllers, which

[20]https://docs.spring.io/spring-framework/reference/web/webmvc.html

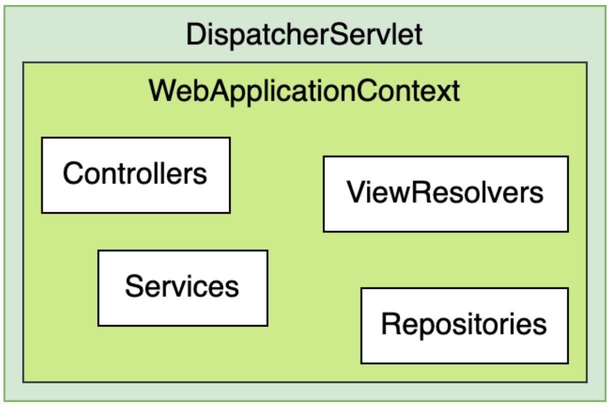


Figure 3.4: Spring WebApplicationContext inside DispatcherServlet containing Spring components (Spring documentation)

define the API and thus build the front door of the application. Secondly, the logic layer holds all the business logic in so-called services. Lastly, the persistence layer called repositories stores and retrieves data with some kind of database. In the context of the QDAcity backend, only the Spring controllers are used at the moment and combined with legacy business logic and data access via Objectify[21].

3.2.5 Spring Security Filter Chain

Spring Security[22] is an authentication and access-control framework, which delivers authentication and authorization mechanisms and protection against exploits. It provides solutions for reactive and servlet[23] applications like the QDAcity backend.

Such applications receive their workload via a servlet mapping from the web server and every request goes through multiple filters before reaching the application itself. These filters contain the DelegatingFilterProxy loading filter beans from the ApplicationContext and forwarding requests to them (see figure 3.5). A specialized Spring filter is the FilterChainProxy, which manages all SecurityFilterChains, registered in the ApplicationContext. Most applications need more than one SecurityFilterChain because different parts of an API require different security configurations (e.g., with and without authentication). Thereby, the SecurityFilterChain contains filters for request mapping, different authentication mechanisms (e.g., email-password, OAuth 2.0, SAML), authentication providers (e.g., Google, Facebook, Twitter), and authorization mechanisms for accessing the user data and checking permissions for certain requests. Additionally, some security filters protect the application against cyber attacks such as the cross-site

[21] https://github.com/objectify/objectify/wiki
[22] https://spring.io/projects/spring-security
[23] https://docs.spring.io/spring-security/reference/5.7/servlet/architecture.html

request forgery (CSRF) attack (Alexenko et al., 2010), where an attacker forces the user to execute malicious operations in the authenticated environment.

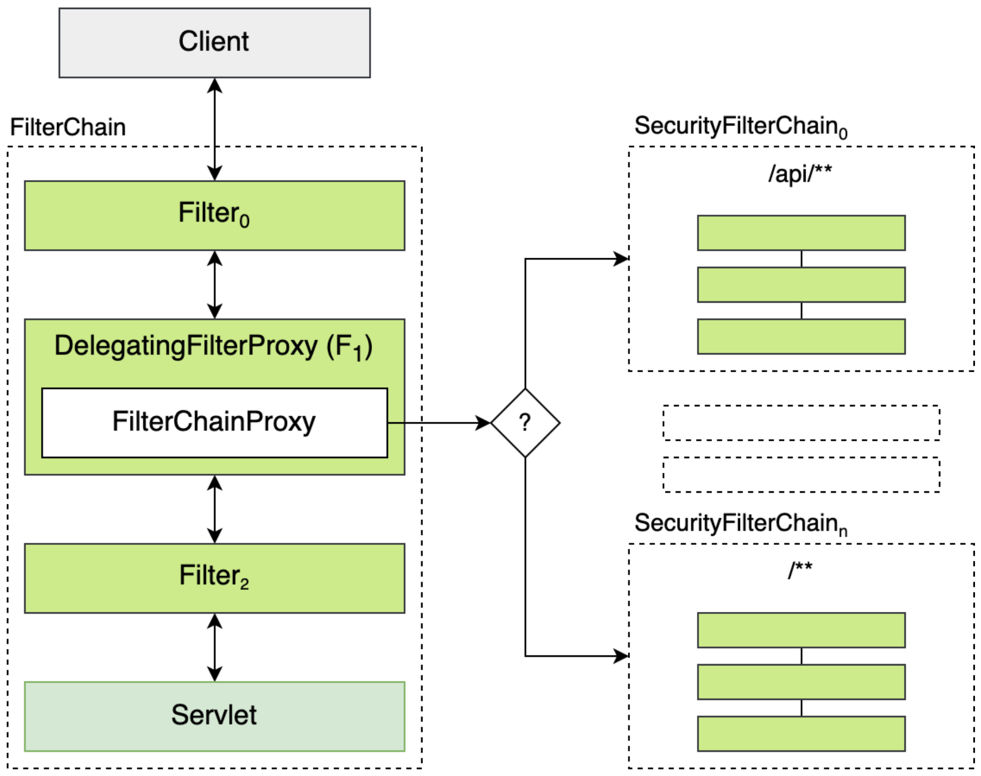


Figure 3.5: Spring SecurityFilterChains receiving requests from the FilterChainProxy and DelegatingFilterProxy inside the FilterChain before reaching the DispatcherServlet (Spring documentation)

The architecture of the Spring authentication mechanism itself starts right at the filter level. Figure 3.6 visualizes the role of the AbstractAuthenticationProcessingFilter within the SecurityFilterChain. When a request is made, the filter intercepts it and triggers the authentication process. The Authentication object, which contains user credentials, is passed to the AuthenticationManager for verification. If authentication fails, control is passed to the failure chain, where multiple handlers process the failure response. In case of success, the flow proceeds with steps that include session management and notification mechanisms.

Figure 3.6 also presents the relationship between the ProviderManager and the AuthenticationManager, which may delegate authentication requests to one or more ProviderManager instances. Each ProviderManager contains a set of authentication providers responsible for validating credentials against different data sources or mechanisms (e.g., lightweight directory access protocol (LDAP), in-memory, etc.). The ProviderManager instances are structured hierarchically under a parent AuthenticationManager, ensuring flexibility and modularity in handling various authentication scenarios.

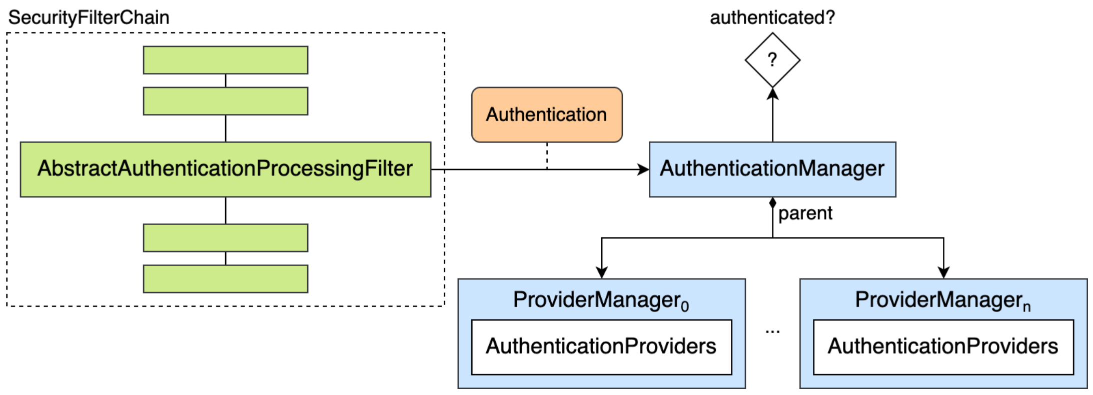


Figure 3.6: Spring AuthenticationManager receiving an Authentication object from the AbstractAuthenticationProcessingFilter and delegating it to ProviderManagers and AuthenticationProviders for authentication (Spring documentation)

Figure 3.7 highlights the SecurityContextHolder, which plays a crucial role in storing security-related information. At the core of this object is the SecurityContext, which contains the Authentication object. This object holds key data about the user, including the principal (the authenticated user), credentials (in this context a JWT), and authorities (the user's roles or permissions). This structure ensures that the current user's authentication details are available throughout the application and that authorization checks can be performed seamlessly.

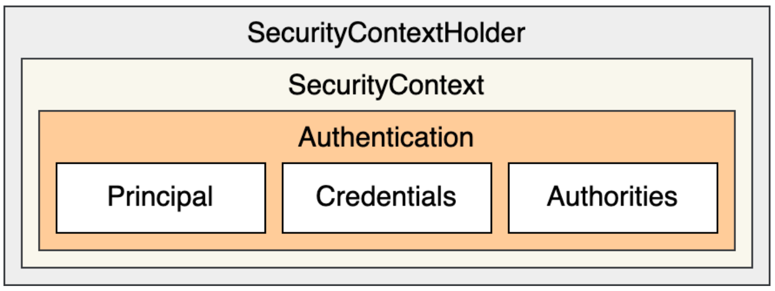


Figure 3.7: Spring SecurityContextHolder containing the SecurityContext and Authentication class (Spring documentation)

The QDAcity Backend uses multiple SecurityFilterChains for different authentication configurations of certain endpoint classes. Some endpoint operations omit authentication, while others, for example, require special authentication providers. All endpoints requiring authentication verify the client's rights via ID tokens in the common JWT format. These tokens contain information about the token type, the issuer, the authorization network, and much more. There are three types of tokens, which are accepted. At first, QDAcity allows Google

ID tokens, checks them via OAuth 2.0, and returns newly generated QDAcity ID tokens, which are the second category and widely used in the application. Lastly, the anonymous ID tokens issued by QDAcity also provide access to some endpoint operations. This restriction of anonymous tokens on many endpoint operations is already some sort of access control in the SecurityFilterChain but on a very coarse level. The more precise authorization mechanism inside the endpoint operations decides based on internal user data if necessary.

This hybrid solution for the QDAcity authorization guarantees back-compatibility to the legacy access control while introducing the more Spring-conform design via the SecurityFilterChain in the first step (see chapter 6.2.2).

3.2.6 OpenAPI Specification

The OpenAPI specification[24] is a standardized description format for REST APIs. It is independent of programming languages and frameworks and defined by the OpenAPI Initiative[25]. These specifications are written in JSON or YAML, but JSON is less resource-consuming as it allows removing unnecessary spaces, tabs, and line breaks (unlike YAML) and thus compressing its size.

In the setting of QDAcity, a SpringDoc-OpenAPI plugin generates the OpenAPI version 3.0 JSON based on the backend Java code during the Maven build process and bundles it into the application. This generated specification acts as a formal definition and documentation of the QDAcity API and its API versioning.

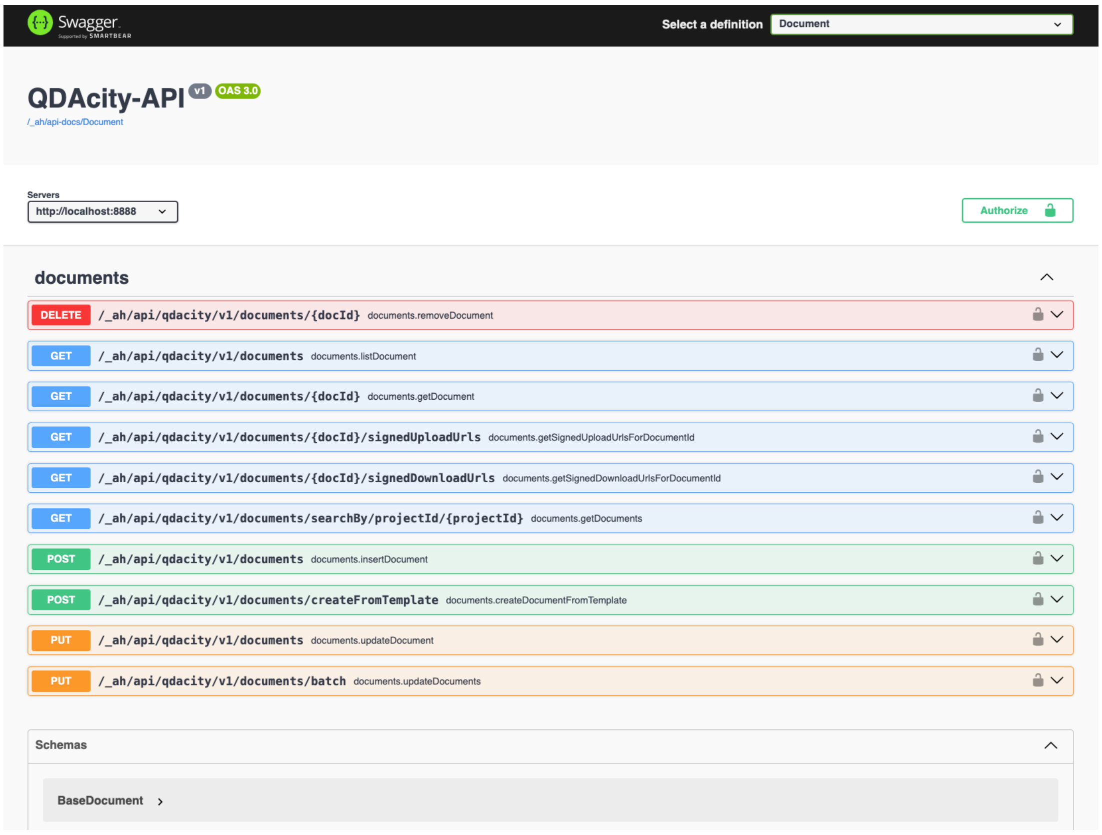
3.2.7 SwaggerUI and API Docs

The QDAcity backend also hosts a visual documentation UI called SwaggerUI[26], which is an HTML web page (see figure 3.8). This page contains multiple sub-pages for every endpoint class of the backend and describes its methods in detail. It, for example, lists the HTTP query parameters, the headers, the request body, and the response code and body. In addition, the SwaggerUI provides a feature allowing developers to try it without the need for writing any frontend client code. They can execute requests on the QDAcity API with arbitrary inputs and receive the corresponding responses without installation.

Swagger provides another feature called API docs, which are endpoints for downloading the underlying OpenAPI specifications. These endpoints allow filtering for every defined group (set of endpoint operations) or a specific API version.

The SwaggerUI and API docs come with the SpringDoc-OpenAPI dependency

[24]https://swagger.io/docs/specification/about/
[25]https://www.openapis.org/about
[26]https://swagger.io/tools/swagger-ui/



QDAcity-API

/_ah/api-docs/Document

Servers

http://localhost:8888

documents

DELETE	/_ah/api/qdacity/v1/documents/{docId}	documents.removeDocument
GET	/_ah/api/qdacity/v1/documents	documents.listDocument
GET	/_ah/api/qdacity/v1/documents/{docId}	documents.getDocument
GET	/_ah/api/qdacity/v1/documents/{docId}/signedUploadUrls	documents.getSignedUploadUrlsForDocumentId
GET	/_ah/api/qdacity/v1/documents/{docId}/signedDownloadUrls	documents.getSignedDownloadUrlsForDocumentId
GET	/_ah/api/qdacity/v1/documents/searchBy/projectId/{projectId}	documents.getDocuments
POST	/_ah/api/qdacity/v1/documents	documents.insertDocument
POST	/_ah/api/qdacity/v1/documents/createFromTemplate	documents.createDocumentFromTemplate
PUT	/_ah/api/qdacity/v1/documents	documents.updateDocument
PUT	/_ah/api/qdacity/v1/documents/batch	documents.updateDocuments

Schemas

BaseDocument

Figure 3.8: QDAcity SwaggerUI sub-page showing the DocumentEndpoint

and are mostly pre-configured. Therefore, the web page and endpoints produce hardly any costs while helping developers understand the QDAcity API and its behavior.

3.3 Frontend

This chapter provides a coarse software architecture of the QDAcity frontend, focusing particularly on its interaction with the backend through Swagger Client. The architecture is designed to deliver a responsive and dynamic user experience, utilizing React's component-based structure to manage the user interface efficiently. Data retrieval from the backend is handled via Swagger Client. This technology ensures consistent and reliable communication between the frontend and backend, facilitating the exchange of data necessary for the application's functionality. The chapter details the key architectural components (see chapter 3.3.1) and the integration of Swagger Client (see chapter 3.3.2), highlighting how these elements work together to achieve a robust and maintainable frontend architecture.

3.3.1 Frontend Architecture

The architecture for the QDAcity frontend involves several distinct components interacting to manage API requests and responses with a focus on handling tokens and interfacing with the QDAcity API (see figure 3.9).

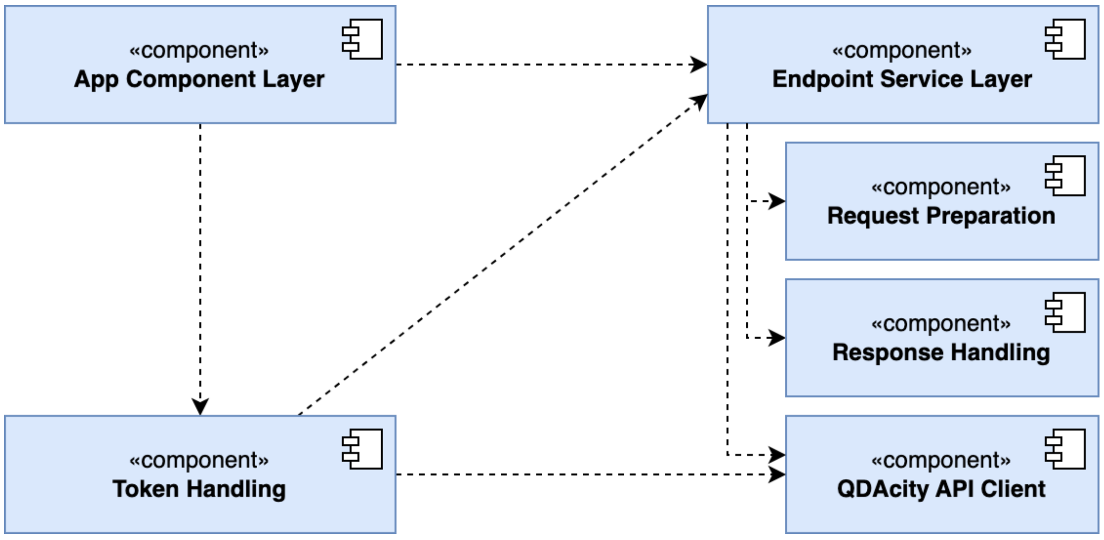


Figure 3.9: Partial UML component diagram of the QDAcity frontend

At the top of this architecture, the app component layer contains the React components and functions as the primary initiator for API interactions. When an API call needs to be made, this component communicates with the endpoint service layer, which acts as a bridge between the application and the various other components involved in preparing and handling API requests. The endpoint service layer is responsible for ensuring that the necessary steps for preparing the requests and handling responses are correctly managed, ensuring smooth communication with the QDAcity backend.

The request preparation component, connected to the endpoint service layer, plays a crucial role in constructing API calls. This component ensures that requests are formatted correctly, taking necessary parameters, headers, and authentication requirements into consideration before an API call is dispatched. After the request is prepared, the QDAcity API client component, which holds the Swagger Client, performs the actual communication with the QDAcity backend.

Once the backend has responded to a request, the response handling component takes over, processing and interpreting the incoming response data. This component ensures that the data is correctly parsed and any potential errors are managed before the response is passed back to the app component layer for further use within the application.

A notable feature in this architecture is, how tokens are managed. The token handling component is dedicated to fetching and managing authentication tokens. It communicates with the endpoint service layer to retrieve tokens when needed and ensures that the API requests include the correct authorization header value. Therefore, the token handling component sets the token directly within the QDAcity API client, ensuring that each outgoing request is authenticated before being sent to the backend.

Thus, the architecture is structured to manage both the flow of data between the QDAcity frontend and backend and the crucial task of token-based authentication.

3.3.2 Swagger Client

Swagger Client is a Node package of the QDAcity frontend that fetches and resolves the OpenAPI JSON for interactions with the QDAcity API. It is called a dynamic client because it does not directly implement a JavaScript function for each endpoint method, but rather gathers the logic from the OpenAPI specification. This generic API client utilizes information about the request URL, parameters, body, authentication, and authorization for its generic client implementation at runtime. Therefore, Swagger Client does not support strictly typed response bodies, which is compatible with the legacy QDAcity frontend, as it did not implement explicit types before the migration.

There are several different programming styles available for the SwaggerClient class. It allows directly using its static functions for preparing and executing requests, but it also provides the possibility to use an instance of SwaggerClient. Thus, the QDAcity frontend uses the instance version for sharing one QDAcity API client across the whole web application.

Furthermore, the frontend structures the API operations via endpoint services analogous to the backend. After calling the backend, these services handle the responses by another central logic, as before with GAPI Client. This logic needed a complete refactoring for the new API client (see chapter 4.2).

3.4 Integration and Deployment

This chapter provides a comprehensive overview of the deployment process for the software artifacts at an architectural level. It details how the system components are packaged (see chapter 3.4.1), distributed, and deployed across various environments (see chapter 3.4.2). The deployment strategy is designed to ensure seamless integration, scalability, and reliability with an emphasis on automating and streamlining the process through the use of modern DevOps practices and tools (see chapter 3.4.3). The chapter offers a clear understanding of how the architecture supports efficient and consistent deployment across different environments.

3.4.1 Software Artifacts

The components of QDAcity map to the corresponding software artifacts as illustrated through their colors in the manifestation diagram of figure 3.10.

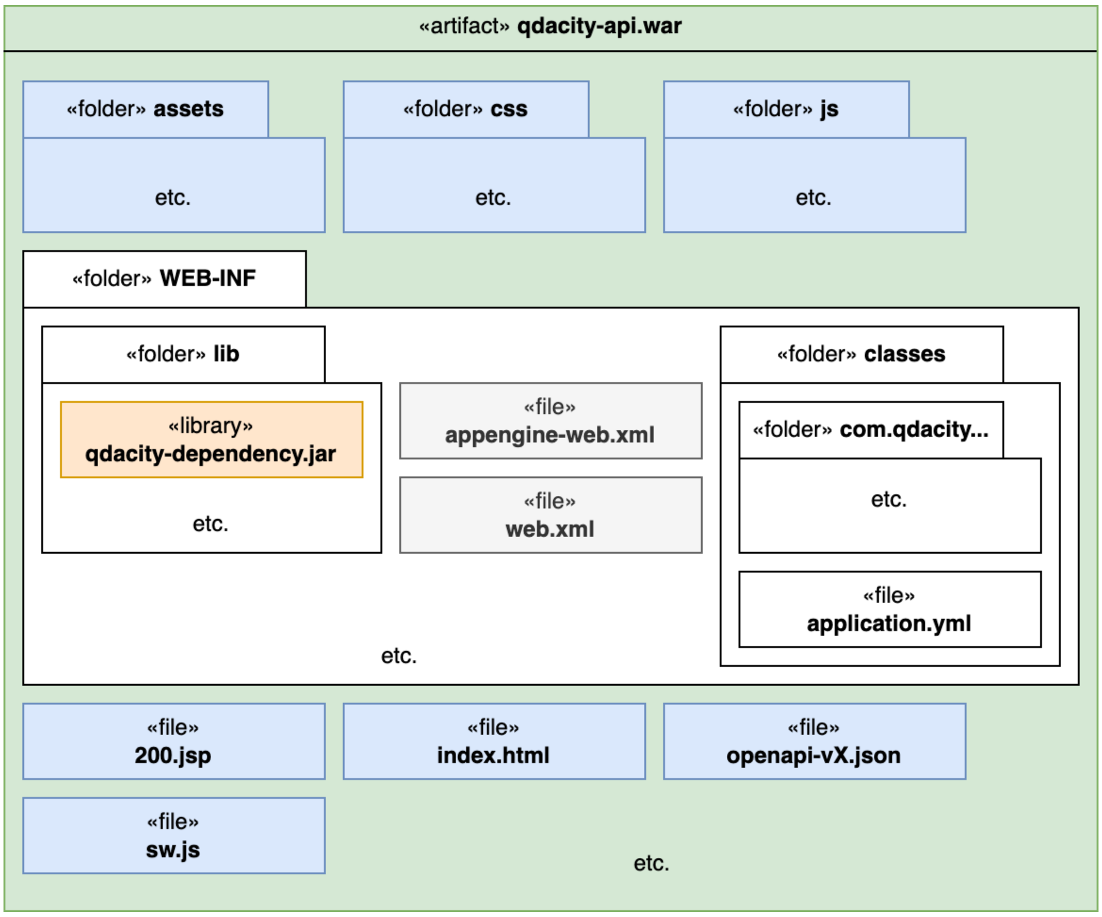


Figure 3.10: UML-oriented manifestation diagram of the qdacity-api.war except for the component mappings

The provided diagram represents the structure of the qdacity-api.war artifact, which is a web application archive (WAR) file used to package the QDAcity backend and frontend for the deployment (see next chapter 3.4.2). The diagram outlines the different files, folders, and libraries, which make up this WAR file. At the highest level, the qdacity-api.war archive includes several folders and files.

The folders "assets", "css", and "js", which are typical in web applications, contain various static resources (e.g., images, style sheets, and JavaScript files), which are utilized by the QDAcity frontend. Moreover, the label "etc." indicates that more specific files would be present inside these directories, but are not visualized.

The core of the QDAcity backend is found within the WEB-INF folder, which is a standard WAR directory for holding configuration files and resources, which should not be directly accessible via the web. Within WEB-INF, there are two key sub-directories, "lib" and "classes". The lib folder contains Java libraries, which the application depends on. An example of these libraries is the qdacity-dependency.jar file, which provides the legacy QDAcity core module. Whereas, the classes folder contains the compiled Java classes of the QDAcity-API module based on Spring Boot. Inside this folder, there is a sub-directory named "com.qdacity...", which represents the Java package structure of the QDAcity-API module. Another file within the classes folder is the application.yml, which holds configuration settings for the backend application.

Additionally, inside the WEB-INF folder, there are further important configuration files like the appengine-web.xml and web.xml. The appengine-web.xml file is used for configuring applications that run on Google App Engine, while web.xml defines servlet mappings, security settings, and other web application configuration details.

Outside of the WEB-INF folder, the diagram depicts several other standalone files within the root of the WAR archive. These include the 200.jsp, index.html, openapi-vX.json, and sw.js. The 200.jsp file is a Jakarta server pages (JSP) file used to dynamically generate HTML content. The index.html file serves as the main entry point for the web application, while the openapi-vX.json file represents an OpenAPI 3.0 specification, which defines the API endpoints of QDAcity. Lastly, the sw.js file is the service worker script, which controls the caching and background tasks of the frontend.

In summary, the manifestation diagram represents the contents and structure of the qdacity-api.war file, detailing its resources, configuration files, libraries, and classes that enable together the deployment and functionality of the QDAcity frontend and backend.

3.4.2 Cloud Infrastructure

As already mentioned, QDAcity deploys to the Google App Engine Gen 2 standard environment for Java with deprecated Gen 1 services but aims a migration to native App Engine Gen 2 in the future (see chapter 6.1.2). However, the CES deploys to the Google Cloud Run[27] environment, which is an alternative solution to App Engine for container-based applications.

[27]https://cloud.google.com/run/docs/overview/what-is-cloud-run

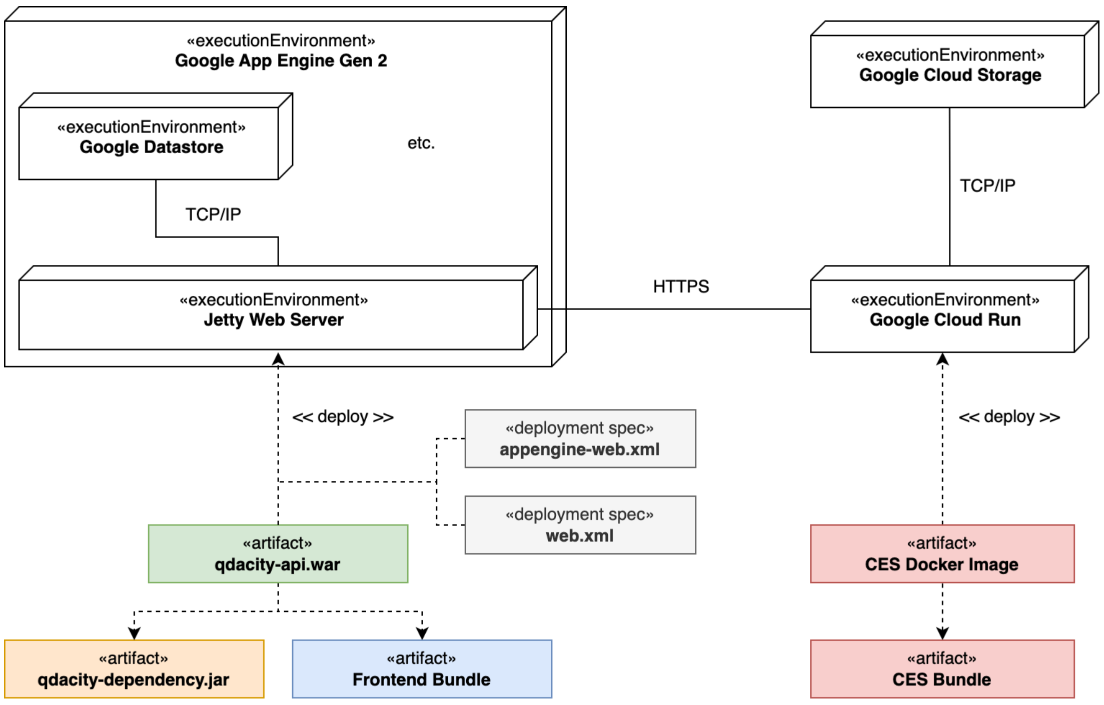

«executionEnvironment» Google App Engine Gen 2
«executionEnvironment» Google Datastore
etc.
TCP/IP
«executionEnvironment» Jetty Web Server
HTTPS
«executionEnvironment» Google Cloud Storage
TCP/IP
«executionEnvironment» Google Cloud Run
<< deploy >>
«deployment spec» appengine-web.xml
«deployment spec» web.xml
«artifact» qdacity-api.war
<< deploy >>
«artifact» CES Docker Image
«artifact» qdacity-dependency.jar
«artifact» Frontend Bundle
«artifact» CES Bundle

Figure 3.11: UML deployment diagram of QDAcity

According to Google Cloud[28], operations experts distinguish three cloud computing service models called infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). These service models differ in their depth of service level. IaaS provides a managed infrastructure containing virtualization and the underlying hardware while a sub-category of IaaS called containers as a service (CaaS) delivers an operating system (OS) for containers on top (e.g., Docker[29]). PaaS automates some parts of the build and deploy process via additional runtime services and SaaS provides a fully configured application (e.g., QDAcity), including the code itself. The Google App Engine[30] is a PaaS model because its standard environment runs Java applications automatically on its internal OS platform and delivers runtime services. Whereas, the Google Cloud Run environment is classified as a CaaS solution.

QDAcity uses the cloud services Google Datastore as its database system, Google Logging, Google Metrics, and many more. It also configures a local devserver and different instance classes[31] of the Google App Engine for the testing and production stages. For example, all students working on QDAcity get their own App Engine instance of class F1, which scales automatically and prevents most

[28]https://cloud.google.com/learn/paas-vs-iaas-vs-saas
[29]https://www.docker.com
[30]https://cloud.google.com/appengine/docs/standard
[31]https://cloud.google.com/appengine/docs/standard#instance_classes

costs. Whereas, the production stage uses a paid variant like B2, which provides manual scaling and more virtual resources while it preserves the low costs of the legacy backend.

Lastly, the before mentioned (see chapter 3.4.1) artifacts are all, except the CES bundle, packaged into the final QDAcity-API WAR and deployed to the App Engine standard environment as seen in figure 3.11. The CES bundle is packaged into a Docker image, executed as a Docker container via Cloud Run, and uses the mentioned Google Cloud Storage (see chapter 3.1.1).

3.4.3 CI/CD Pipeline

QDAcity versions its code base via Git on GitLab[32] and additionally uses the so-called GitLab CI[33]. This is a tool for continuous integration and continuous delivery[34] (CI/CD), which regularly integrates, delivers, and even deploys the QDAcity application and triggers itself automatically for every commit and merge.

The CI/CD pipeline of QDAcity is based on the GitLab CI and has four major stages. At first, it builds the project artifacts, secondly tests the code, then deploys the current build to a configured test environment, and finally releases a new version to production. Of course, not every triggered pipeline run performs the whole procedure, but rather skips the deployment for feature branches and disables releases in the fork projects for students. Moreover, most jobs of each stage work in a parallel setup while the stages mostly imply a sequential order.

Starting with the backend build job, on the one hand, the pipeline configures the QDAcity backend via Gulp[35] and executes the Maven build process. The frontend build procedure, on the other hand, combines multiple jobs. At first, it packages an abstract frontend bundle. This intermediate variant is used by several test jobs and later in the test stage interpolated with a production configuration, resulting in an environment-specific bundle. The deploy job lastly copies the bundle to the final WAR artifact containing the QDAcity frontend and backend. Consequently, the last missing job of the build stage bundles the CES separately.

The subsequent test stage runs the unit tests for the application components, the frontend translation tests, Lighthouse[36] tests for the UI, and the Selenium[37] acceptance test. In addition to the tests, it also performs formatting checks, linting, the aforementioned API configuration, and an API version check.

[32]https://gitlab.com
[33]https://docs.gitlab.com/ee/ci/
[34]https://about.gitlab.com/topics/ci-cd/
[35]https://gulpjs.com
[36]https://developer.chrome.com/docs/lighthouse
[37]https://www.selenium.dev

At the end of the fully automated pipeline remains the deploy stage, which delivers the current QDAcity version to a configured test instance of the Google App Engine. This test environment allows the product owners to check the application manually as a last quality assurance.

The final release stage is only semi-automated because it requires a manual trigger to deploy the application to the production environment. That enables the developers to decide carefully if a release makes sense or even not.

All in all, the usage of an automated pipeline for development, testing, and operation activities (i.e., DevOps tasks) ensures high-quality standards, running software, and collaboration between many developers.

3.5 Technology Decisions

This chapter outlines the critical decisions made in the selection of technologies. It discusses the rationales behind choosing the Spring Boot framework (see chapter 3.5.1), the Jetty web server (see chapter 3.5.2), the SwaggerUI (see chapter 3.5.3), and the Swagger Client (see chapter 3.5.4) in detail, considering factors such as performance, maintainability, and compatibility with existing systems. By documenting these decisions, this chapter provides transparency into the architectural process, offering insights into how the chosen technologies effectively contribute to the overall system. All topics introduce the alternatives and show their commonalities and differences with the selected variant.

3.5.1 Backend Framework

The most important question at the beginning of this thesis was, what framework should QDAcity choose for its backend? There were two options discussed before making the final decision which are presented in the following part.

The Dropwizard[38] framework is a bundle of many renowned Java libraries in the context of web applications, but it does not implement bigger solutions on its own. However, it provides ready-to-run configurations reducing the time-to-market and maintenance overhead. As described in chapter 3.2.2, Spring Boot configures the Spring framework and other Spring modules by default resulting in a production-ready setup. However, unlike Dropwizard, the Spring framework and Spring modules implement their approaches, architectures, designs, and logic for certain problems and additionally combine them with well-known libraries (see table 3.2). They also support more options for the same feature and thus enable their users to select the best fitting or most liked solution. For example, users

[38]https://www.dropwizard.io/en/stable/getting-started.html#overview

can take the HTTP web server Jetty, Tomcat, Undertow, or many others.[39] The same holds for the logging of the application where Spring provides a so-called commons logging bridge that recognizes Log4j or SLF4J API implementations in its context or uses Java's logging by default.[40]

Feature	Spring Boot	Dropwizard
HTTP Webserver	Jetty, Tomcat, etc.	Jetty
Web Framework	Spring (Web) MVC	Jersey
Auth Framework	Spring Security	Jersey Security
Dependency Injection	Spring DI	Eclipse's HK2
JSON-Object Mapping	Jackson	Jackson
Data Validation	Hibernate Validator	Hibernate Validator
(Relational) DB Access	Spring Data	JDBI and Hibernate
Logging	Log4j and SLF4J Bridge	Logback (SLF4J)
Observability	Micrometer	Metrics

Table 3.2: Comparison of Spring Boot and Dropwizard features based on technologies according to the Spring and Dropwizard documentations

Unfortunately, the performance of these frameworks is not comparable in the scope of this thesis, because both possess a high modularity of the dependencies and an even higher flexibility in terms of their technology stacks. However, one framework with different technology combinations allows performance comparisons as described in the next chapter 3.5.2 about the web server decision.

The official documentation of Spring and Dropwizard are both well structured, but Spring has many community pages such as Baeldung[41], which deliver additional articles and tutorials for developers. This helps with learning the fundamentals of Spring Boot and setting things up.

According to GitHub[42], Spring Boot has much more stars and forks, but both release frequently. Additionally, Dropwizard heavily depends on third-party projects, which can affect the framework and thus the application as well. Whereas, Spring Boot uses a lot of dependencies where many of them also are Spring modules, which are patched frequently. Consequently, both frameworks deliver long-term support, while Spring Boot depends less on third-party code.

All in all, Spring Boot covers more flexibility, the same or even more features, and a much broader community than Dropwizard. Thus, Spring Boot was chosen

[39] https://docs.spring.io/spring-framework/reference/web/websocket/server.html#websocket-server-deployment
[40] https://docs.spring.io/spring-framework/reference/core/spring-jcl.html
[41] https://www.baeldung.com
[42] https://github.com

and the selected version 2.7.18 heavily depends on Java 8, as already described in chapter 3.1.3.

3.5.2 Web Server as Servlet Container

The Spring Boot framework requires a servlet container, more commonly known as a web server, in which it can run the DispatcherServlet (see chapter 3.2.4). It supports many different web servers, which all implement the necessary Jakarta specifications (e.g., Jakarta servlet).

A comparison of three selected Spring Boot servlet containers from Baeldung[43] measures their performance and efficiency according to some startup and request metrics against a Spring Boot setup (see tables 3.3 and 3.4). These three setups with a Tomcat, Jetty, and Undertow web server perform comparably in all metrics collected by the Spring Boot Actuator and Apache Bench. Jetty requires less memory, loads fewer classes, and needs fewer threads than Tomcat and Undertow, but all three are comparably efficient. However, Undertow can handle more requests at the same time than Tomcat and Jetty, but no setup is clearly better. These results refer to Spring Boot version 3.1.5 and recent webserver versions, while the current setup of QDAcity contains an older Spring Boot 2.7.18 with an older web server version (see chapter 3.1). However, QDAcity is aiming to upgrade these technologies as part of the future switch to native App Engine Gen 2 and the latest long-term support (LTS) version of Java.

Startup Metric	Tomcat	Jetty	Undertow
jvm.memory.used (MB)	168	**155**	164
jvm.classes.loaded	9869	**9784**	9787
jvm.threads.live	25	**17**	19

Table 3.3: Comparison of Spring Boot setups with Tomcat, Jetty, and Undertow according to some startup metrics collected via Spring Boot Actuator (Baeldung article)

Benchmark Metric	Tomcat	Jetty	Undertow
Requests per second	1542	1627	**1650**
Average time per request (ms)	6.483	6.148	**6.059**

Table 3.4: Comparison of Spring Boot setups with Tomcat, Jetty, and Undertow according to some benchmark metrics collected via Apache Bench (Baeldung article)

Furthermore, the configurations of these setups vary because Spring Boot ships an embedded Tomcat web server by default, while Jetty and Undertow require

[43]https://www.baeldung.com/spring-boot-servlet-containers

an explicit substitution of this dependency. In addition, Google App Engine Gen 2 with bundled Gen 1 services provides a pre-configured and automatically managed Jetty of version 9, which the legacy QDAcity application before used. Therefore, it was possible to adapt the legacy configurations of this solution. Although Google App Engine Gen 2 does not ship a Jetty webserver, it is possible to switch to the embedded variant and the latest version in the future.

Even if Spring supports several web server implementations, they still differ in their behavior or configuration for certain features. Therefore, the list is limited to only three options that are fully compatible and open source. Each of these web servers possesses over 1000 stars, more than 100 forks, and hundreds of versions according to their GitHub repositories. Consequently, all options guarantee long-term support and regular updates.

All in all, Jetty delivers a comparable or even better performance and efficiency than Spring's choice Tomcat and the alternative Undertow. QDAcity chose the Jetty for the legacy setup and Google App Engine Gen1 services ship it. Thus, the provided Jetty web server of version 9 was taken as the Spring Boot servlet container for the QDAcity backend.

3.5.3 API Client for Manual Testing

The legacy QDAcity backend based on the Google Endpoints framework delivered automatically a so-called API explorer, which was hosted directly by the App Engine. It allowed the developers to test the endpoint operations provided by the QDAcity API. With the migration to the Spring Boot framework, this comfortable tool was lost and needed a successor.

In the context of Spring, there are many possibilities, but the evaluation of possible options was limited to Postman[44], Insomnia[45], and SwaggerUI as common representatives. Postman and Insomnia are local client applications for testing APIs manually, whereas SwaggerUI is a hosted web application reached via the web browser. Both variants help to perform HTTP requests against an API via automated processes, state, and a self-explanatory user interface. The locally installed clients allow importing an OpenAPI specification for an initial setup of a so-called collection. This is the place, where all metadata and request data is stored. It is possible to export these collections partially or completely for sharing with other developers. SwaggerUI is directly generated based on an OpenAPI specification and provides REST endpoints for downloading the underlying documents.

The most important difference in terms of usability for developers comes from

[44]https://www.postman.com
[45]https://insomnia.rest

the initial concept of these tools. A local installation of desktop applications like Postman and Insomnia requires more time and causes many different settings compared to a centrally deployed web app like SwaggerUI. Moreover, SwaggerUI can also function as technical documentation for an API, which is potentially designed as a product (API as a product). In this case, it can be deployed for multiple API versions on multiple paths.

Another difference between these options stems from their pricing model. Postman and Insomnia are paid products with customer support and special features (e.g., team collaboration) for charged versions, but also provide a free plan for local usage with a reduced service frame. SwaggerUI is a permissively licensed open source project and thus free of charge, well known, and frequently released according to its GitHub repository. However, this is only the half-truth as a web application needs to be hosted somewhere causing costs as well. Lastly, all options guarantee long-term support and regular updates, but for a different price.

All in all, SwaggerUI delivers a centrally managed tool for developers to test the QDAcity API while also functioning as an online documentation, which can be versioned. In addition, it is open source and libraries such as SpringDoc-OpenAPI help integrate it into the Spring Boot framework. Thus, the decision was ultimately made to use SwaggerUI 5.11.8 installed and configured via SpringDoc-OpenAPI version 1.8.0. This was the latest version supporting Spring Boot major version 2 at that moment.

3.5.4 Frontend API Client

The QDAcity frontend previously used the GAPI client, which allowed accessing the legacy backend based on Google Endpoints. Therefore, it loaded and resolved the discovery document describing the QDAcity API and secondly handled the JWT token for authentication and authorization. This discovery document is a specification for Google Endpoints APIs, but the migrated QDAcity backend now relies on Spring Boot and the independent OpenAPI specification (see chapter 3.2.6). Consequently, it was necessary to replace the GAPI Client, so that it can use this new OpenAPI specification.

There were three different options evaluated for the new API client. The first option is Swagger Codegen[46], which generates JavaScript code for the client and server stubs for frontend testing based on the OpenAPI document. Whereas, the dynamic clients Swagger Client and OpenAPI Client Axios[47] do not create a client code base, but rather load and resolve the OpenAPI specification at runtime like the GAPI Client with the discovery document before. These two dynamic client options differ on the underlying HTTP client, which they use. Swagger Client

[46]https://swagger.io/tools/swagger-codegen/

[47]https://www.npmjs.com/package/openapi-client-axios

directly uses the fetch implementation of Node, whereas OpenAPI Client Axios builds on the Axios[48] client for JavaScript.

While both approaches fulfill the same task, accessing the API in the frontend, the static variant via a code generator like Swagger Codegen needs no startup or preparation procedure at runtime. It does not load any specification document but gets directly shipped with the frontend JavaScript code. Whereas, this generated client must be versioned or integrated into the the build process, which requires a more complex setup compared with a generic client code that loads a predefined OpenAPI specification at runtime.

From a maintenance point of view, the Swagger GitHub projects (i.e., Swagger Codegen and Swagger Client) receive more attention from the community and are also released more often according to the node package manager (NPM) registry. Thus, the solutions provided by Swagger ensure long-term support and frequent updates.

All in all, Swagger Client provides the best compromise of functionality, performance, setup complexity, and support for the QDAcity frontend. The selected version 3.28.8 of Swagger Client was the latest version during its integration into the QDAcity frontend.

3.6 Product Decisions

This chapter focuses on key architectural decisions that have influenced the development of the product. It starts with an overview of the Maven multi-module setup for the project (see chapter 3.6.1). Following this, the authentication providers used in the system are discussed (see chapter 3.6.2), explaining the rationales behind their selection to ensure secure user access and authorization. The management of API breaking changes is then examined (see chapter 3.6.3), including the strategies employed to minimize disruption and maintain compatibility. Lastly, the structure of sub-pages within SwaggerUI is outlined (see chapter 3.6.4), highlighting how API documentation is organized for optimal accessibility and ease of use. These decisions shape the product's technical foundation and its ability to scale and adapt over time.

3.6.1 Maven Multi-Module Setup

The QDAcity backend uses Maven to manage its dependencies like the Spring Boot modules via a single file called the POM. Maven provides project setups with one single module or multiple sub-modules, which is then called a multi-module setup.

[48]https://axios.rest

At first, a proof of concept (PoC) for the single module setup was performed. It showed that the legacy backend required a dependency called Datanucleus, which has a transitive dependency called ASM Enhancer. This transitive dependency is also included in Spring Boot major version 2, but in a later version. Unfortunately, Spring Boot is not compatible with the version contained by Datanucleus major version 3. Moreover, Google App Engine Gen 2 with bundled Gen 1 services supports only this version of Datanucleus, so it prevents an update.

Consequently, the second approach was a multi-module setup of Maven with a parent module and two sub-modules, one for the legacy code and one for the Spring Boot backend code. Thereby, the second sub-module named QDAcity-API depends on the legacy sub-module called QDAcity and both inherit the configurations of the parent module named QDAcity-Backend. This solution worked out and can be extended in the future by additional sub-modules or refactored back to a single-module setup (see chapter 6.1.5).

3.6.2 Authentication Providers

The legacy authentication logic implemented a JWT token validator in the backend for QDAcity, Google, Facebook, and Twitter (now "X") ID tokens, but used only the QDAcity and the Google variant at that moment. Thus, the question arose of which authentication providers should remain for the QDAcity backend, during the refactoring of the authentication mechanism to Spring Security.

Therefore, the authentication mechanism of the Spring Boot backend accepts only the QDAcity and the Google ID tokens.

3.6.3 API Breaking Changes

One goal during the migration to Spring Boot was to avoid breaking changes in the QDAcity API. In the following cases, changes were necessary due to inconsistencies, path matching (or routing), refactored token headers, and OpenAPI standards.

First of all, the QDAcity endpoints in Spring Boot commonly define their sub-path segment named like the classes themselves. During the migration, this rule was enforced with one exception. For technical reasons, the analytics endpoint contains an operation that does not map via the path "analytics" but via "system".

Moreover, each endpoint operation needs a unique path pattern that can be matched by the Spring SecurityFilterChain. This was already discussed before in chapter 3.2.5. This rule caused some changes in the QDAcity API as well.

The Spring Security setup of the authentication at the QDAcity backend uses

Spring's OAuth 2.0 validator for Google ID tokens. Thereby, requests should carry only this JWT token as part of the authorization header prepend with the term "Bearer". Consequently, this authentication method is oftentimes called bearer authentication. However, the legacy QDAcity backend needed a string defining the used token type (i.e., QDAcity or Google) and parsed it as a third sub-string from the authorization header value. This is a contradiction and thus the Spring Boot backend now extracts this information directly from the JWT token payload and enforces a valid authorization header value. This new authentication rule changes the behavior of the API and requires the API consumers to remove the token type sub-string.

Another topic enforces some endpoint operations to change their intuitive HTTP method due to OpenAPI major version 3 standards. According to these rules, operations requiring an HTTP body should not use GET and DELETE but rather POST or PUT. In some cases, this was not met by the legacy operations of the QDAcity API and therefore changed. Unfortunately, these new HTTP methods often do not feel intuitive and should get a comment explaining this special situation.

All these breaking changes were necessary or at least valuable for the future of the QDAcity API.

3.6.4 SwaggerUI Sub-Pages

SwaggerUI visualizes all endpoint operations of the QDAcity API sorted at first by their HTTP method and secondly in alphabetic order. Additionally, the endpoint classes bundle their operations inside a collapsible area. The initial setup of SwaggerUI only used these elements for structuring the web page. At some point, SwaggerUI could not show the exemplary objects of the request body while the number of migrated endpoints was growing. However, the OpenAPI specification included all the information and thus it must be a bug of SwaggerUI.

The solution for this issue now only contains one endpoint class per page and splits the SwaggerUI into multiple sub-pages. This structures the tool even more.

4 Design and Implementation

This chapter outlines the design and implementation details of the system. It first addresses the QDAcity backend (see chapter 4.1), which was migrated from the Google Endpoints framework to Spring Boot. Consequently, this chapter will focus exclusively on components affected by the migration. Following this, the frontend is covered (see chapter 4.2) with a strong focus on the components involved in communication with the QDAcity API. Both chapters reflect only the current state and do not mention the changes that occurred during the migration process.

4.1 Backend

The design and implementation of the QDAcity backend gives a detailed overview of the migrated components, configurations, and the usage of the chosen technologies. It starts with the basics in the Spring Boot setup (see chapter 4.1.1) and the web server configuration (see chapter 4.1.2), followed by smaller topics like dependency injection (see chapter 4.1.3), exception handling (see chapter 4.1.4), and data serialization (see chapter 4.1.5). Lastly, it closes with the bigger topics including the authentication mechanism via the SecurityFilterChain (see chapter 4.1.6), the design of the migrated endpoints as wrappers (see chapter 4.1.7), and the API implementation and specification (see chapter 4.1.8).

4.1.1 Spring Boot Setup

The QDAcity backend project is structured as a Maven multi-module project with a setup, designed to manage the build and deployment process through Maven and Google App Engine. At the root level of the project, the parent-pom.xml file orchestrates the configuration for the entire project, containing key information such as the project name QDAcity-Backend and the root-level artifact ID qdacity-be. This parent POM file defines the two core modules of the project: the QDAcity module, located at the root, and the QDAcity-API module, housed within its qdacity-api directory. Additionally, the parent POM specifies Java version 8 as the source and target version and manages dependencies shared across modules, such as Lombok, which is used for automatic code generation across the entire project.

The QDAcity module's build configuration is managed by a pom.xml file that is automatically configured through a Gulp script, which interpolates this file from a pom.base.xml file located at the root. This module has the artifact ID qdacity and inherits from the qdacity-be parent. It uses the maven-jar-plugin to build a separate Java archive (JAR), which is later used as a dependency in the QDAcity-API module. The main functionality of this module is providing the necessary compiled legacy classes and resources, which are imported into the QDAcity-API module for integration with the Spring Boot framework.

Similarly, the QDAcity-API module's pom.xml file is interpolated by Gulp from a base POM located in its directory named qdacity-api. This module has the artifact ID qdacity-api and inherits from the parent project qdacity-be, as well. A key part of its configuration is its use of Spring Boot, managed through the Spring Boot dependencies import that provides centralized version management for various Spring libraries required by the project. Notably, this module includes the QDAcity module, pulling in the dependency JAR file to integrate legacy components into the Spring Boot backend.

The dependencies for QDAcity-API also include essential Spring Boot libraries. However, instead of using the default embedded Tomcat server, the project runs on the Jetty web server provided by Google App Engine Gen 2 with bundled Gen 1 services, reflecting the deployment environment. Several dependencies are upgraded to address security vulnerabilities found in older versions of Spring Boot, particularly version 2.7.18 as described in chapter 3.1.3. The project also utilizes SpringDoc-OpenAPI to generate the OpenAPI specification by automatically extracting API details from annotations in the code.

The QDAcity-API module employs various Maven plugins to configure the build and deployment process. The Maven compiler plugin ensures that the Java version remains 8, and the Spring Boot Maven plugin sets the main class of the application to QdacityApiApplication, allowing the Spring Boot application to launch. Google App Engine deployment is managed by the App Engine Maven plugin, with the configuration set to start the Spring Boot application on Jetty during the pre-integration-test phase. The App Engine environment is essential for both local testing and cloud deployment. The module includes an appengine-web.xml file, located in the WEB-INF directory, which configures the deployment specifics for App Engine.

Additionally, the SpringDoc-OpenAPI Maven plugin automates the generation of the OpenAPI specification during the integration-test phase. After the backend is started by the App Engine Maven plugin, the OpenAPI specification is downloaded from the API docs endpoint path "/_ah/api-docs" and saved into the specified output directory. The resulting OpenAPI JSON file is copied into the root of the WAR package created by the Maven WAR plugin, which repackages the WAR in the post-integration-test phase. This WAR file includes a WEB-INF directory that contains essential configuration files such as appengine-web.xml and web.xml, along with the generated OpenAPI specification.

Maven profiles are used to handle the differences in environment-specific behavior during the OpenAPI generation. There are profiles for Linux, Windows, and MacOS, each automatically activated based on the OS type. The Linux profile enables the App Engine Maven plugin to stop the running backend server after the integration-test phase, while the Windows and MacOS profiles utilize the Exec Maven plugin to run scripts that terminate the Jetty process. These scripts, specific to each OS, ensure that the backend server is correctly shut down after the OpenAPI generation.

4.1.2 Web Server Configuration

As described before, the web server configuration for the QDAcity backend project is tailored to run on Google App Engine Gen 2 with bundled Gen 1 services using Jetty as the web server. This is achieved through the project's pom.xml file, where the Spring Boot Starter Web dependency explicitly excludes the Spring Boot Starter Tomcat dependency. Instead, the Spring Boot Starter Jetty dependency is added with the scope set to provided, indicating that Jetty is supplied by the App Engine environment during deployment.

The QdacityApiApplication class is central to the Spring Boot setup and configured for the startup via the ServletInitializer class. This ensures that the Spring Boot application initializes itself correctly and that the DispatcherServlet, which routes incoming requests to the appropriate endpoint, is properly configured. This setup allows for seamless integration of the Spring Boot framework with the external Jetty server.

The project's web.xml file further defines the servlet configuration. It specifies a welcome file (index.html) and servlet mappings for the DispatcherServlet containing the API base path, the SwaggerUI path, and the API docs path. Additionally, the web.xml file contains a servlet configuration, which serves the frontend during local development, mapping it to the root path, but without conflicting with the other paths. It also ensures that 404 errors in production redirect to the frontend, allowing it to manage non-existent links softly. Another important part of the XML configuration includes the servlet, filter, and listener registrations and mappings. Security is enforced via a security constraint defined in the web.xml, which specifies a transport guarantee of "CONFIDENTIAL". This constraint ensures that communication is encrypted using HTTPS in production environments, though it is not active for local development.

In addition to these XML-based configurations, the project also contains Spring's Java-based configuration approach for registering servlets, filters, and listeners through dedicated configuration classes. The ServletsConfig class, annotated with @Configuration, defines beans that register servlets within the Spring context. Bean methods annotated with @Bean create and configure instances of ServletRegistrationBean, which allows the registration of servlet objects with a specified name, load order, and URL mappings.

Similarly, the FiltersConfig class also uses Spring's configuration to define beans for filter registration. The FilterRegistrationBean instances, created within bean methods, register filters by specifying the filter object, its name, and the URL patterns to which the filter should apply. These filters register an ObjectifyFilter, enabling the Objectify context for the Datastore, and a non-www filter, redirecting requests on the root domain.

Finally, the ListenersConfig class follows the same pattern, providing a Spring-managed configuration for listener registrations. The bean methods in this class create instances of ServletListenerRegistrationBean, which register listener objects.

However, these Java-based configurations are deactivated at the moment and should replace the XML-based configurations for servlets, filters, and listeners after the future migration to an embedded web server and native Google App Engine Gen 2 (see chapter 6.1.2).

4.1.3 Dependency Injection of Legacy Components

Spring Boot's dependency injection mechanism is built on the IoC principle, which manages the lifecycle and dependencies of components known as Beans. These are typically identified by Spring-specific annotations like @Service or @Component, allowing them to be automatically detected and injected as needed. The IoC framework plays a crucial role in decoupling component management from business logic, creating a flexible and maintainable architecture. For a more detailed discussion on how this mechanism works and fits into the overall system design, refer to chapter 3.2.3.

However, legacy components that lack such Spring annotations require a different approach to be integrated into the Spring context. Specifically, a configuration class is necessary to inject these legacy components into the modern system, ensuring they are still available and functional despite not being marked with Spring's annotations. Of course, this approach is also used for the servlet, filter, and listener configurations, mentioned before.

The LegacyBeansConfig class fulfills this role by creating instances of legacy classes, including token validators, which are essential for the authentication mechanism and legacy endpoints used by the endpoint wrappers (see chapter 4.1.7). Through this configuration, older components can be injected and maintained alongside more modern, Spring-compliant ones.

Although this configuration enables the smooth integration of legacy components, it is intended as a temporary measure. In the future, all legacy classes required by other classes, including especially legacy controllers and data access objects (DAOs), should be labeled with Spring annotations that automatically instantiate and inject these classes in the ApplicationContext.

4.1.4 Exception Handling of Legacy Exceptions

In this Spring Boot application, the exception handling mechanism is designed to globally manage specific types of custom exceptions. This is achieved through the GlobalExceptionHandler class, which ensures consistent error responses when certain conditions are violated, such as bad requests or unauthorized access. The class is annotated with @ControllerAdvice, signaling to the Spring context that the exception handling methods within this class apply globally across all endpoint classes, referred to as Controllers in the context of Spring.

The GlobalExceptionHandler class extends ResponseEntityExceptionHandler, a Spring Boot class responsible for handling common response-related exceptions. By extending this class, the custom GlobalExceptionHandler leverages its default behavior while adding custom logic for specific exceptions relevant to the application's domain. The two exceptions addressed in this case are BadRequestException and UnauthorizedException, each of which is handled by a corresponding method.

For the BadRequestException, the method is registered with the @ExceptionHandler annotation, which signals that this method will handle any exception of type BadRequestException. The @ResponseStatus annotation specifies that the HTTP response status should be set to 400 bad request. The method returns a ResponseEntity object, where the response body contains the message extracted from the exception. This allows the API to provide detailed feedback to the client about the nature of the bad request.

Similarly, the UnauthorizedException gets handled comparably. The method responsible for handling this exception is annotated with @ExceptionHandler, telling Spring that the method should be invoked if such an exception is thrown. Additionally, the @ResponseStatus annotation ensures that the HTTP response will return a 401 unauthorized status. Like the previous handler, it constructs a ResponseEntity object with a message extracted from the exception to clarify the cause of unauthorized access.

By adopting this global exception handling approach, the application achieves centralized error management, which decouples exception handling logic from the controller methods. The @ControllerAdvice annotation, in combination with @ExceptionHandler, allows for clean, reusable, and maintainable code, while ensuring that appropriate HTTP status codes and informative messages are consistently returned to the client. This improves the overall robustness of the API.

In summary, the design integrates Spring Boot's robust default exception handling with custom logic, resulting in an API that responds to bad requests with a 400 status and unauthorized access with a 401 status, along with meaningful error messages. This structured approach promotes better usability and debugging for API consumers.

4.1.5 Data Serialization for Compatibility

Serialization and deserialization are processes fundamental to converting objects to and from a format that can be transmitted or stored, such as JSON. In the context of a Spring Boot application, the framework, in conjunction with the Jackson[1] library, automates the serialization and deserialization of objects at the API boundary. This mechanism ensures that objects are converted to JSON when transmitted as responses and that incoming JSON requests are mapped back to Java objects, streamlining API communication.

However, the serialization behavior of the legacy Google Endpoints framework presents notable differences when compared to the default Jackson ObjectMapper used in Spring Boot, particularly in handling Blob and Long values. In the case of Blob objects, the Google Endpoints framework converts them into Base64-encoded strings, whereas Jackson serializes a Blob as an object containing various fields that store the data. Similarly, Long values are serialized as strings in the Google Endpoints framework, while Jackson defaults to representing them as numeric values.

The legacy module of QDAcity contains specific implementations to handle the serialization and deserialization of Blob objects. The ApiBlobSerializer class implements the JsonSerializer interface of Jackson. Within its serialize method, it converts Blob objects into their corresponding Base64-encoded string representation. Conversely, the ApiBlobDeserializer class implements JsonDeserializer and defines a deserialize method that reverses this process. To ensure the proper handling of Blob objects in models or beans that interact with the API, the @JsonSerialize and @JsonDeserialize annotations of Jackson are applied, wherever Blob fields appear in the data models.

For handling Long values, the JacksonConfiguration class customizes the default

[1]https://github.com/FasterXML/jackson/

Jackson configuration in Spring Boot, ensuring that the default serializer for Long-wrapper and primitive Long values is the ToStringSerializer. This configuration overrides the standard numeric serialization of Long values, forcing them to be serialized as strings to maintain compatibility with the QDAcity frontend expecting this format.

4.1.6 Authentication and Token Types

The authentication mechanism in the backend of QDAcity relies on Spring Boot and Spring Security, employing custom JWT tokens and various validators to manage both registered and anonymous user authentications. The architecture revolves around several key classes and interfaces that define the behavior of authentication providers, token validators, and token structures, which are all integrated into Spring Security's filter chains introduced in the architecture chapter 3.2.5.

There are two custom implementations of Spring's AuthenticationProvider interface that handle registered and anonymous user authentication: RegisteredJwt-TokenAuthenticationProvider and AnonymousJwtTokenAuthenticationProvider. As shown in figure 4.1, both implementations use the QdacityApiAuthenticator class to perform the actual authentication. Therefore, these providers implement two methods: supports, which checks whether the incoming authentication object is of type BearerTokenAuthenticationToken, and authenticate, which initiates the authentication process by delegating it to either the authenticateRegistered-JwtToken method or the authenticateAnonymousJwtToken method in the QdacityApiAuthenticator.

The core class responsible for authentication is QdacityApiAuthenticator. It interfaces with legacy code via the ITokenValidator interface, which is implemented by AnonymousTokenValidator, CustomJWTValidator, and GoogleAccessToken-Validator. QdacityApiAuthenticator exposes three primary public methods: authenticateRegisteredJwtToken, authenticateAnonymousJwtToken, and getUser-ByToken. Both authenticateRegisteredJwtToken and authenticateAnonymous-JwtToken are designed to determine whether a request is authenticated or not.

The getUserByToken method is similar but focuses on retrieving a stored user based on the JWT token from the authorization header. It also extracts and validates the token's header and payload. Depending on the authorization network, it uses the appropriate validator to authenticate the token and return the associated user. If neither custom JWT validator is successful, a fallback mechanism using GoogleAccessTokenValidator is employed, which attempts to authenticate the token via Google. This logic plays a key role in the next chapter 4.1.7.

As visualized in figure 4.2, the JWT token used in the authentication process consists of two main parts: the header and the payload. The header contains a field

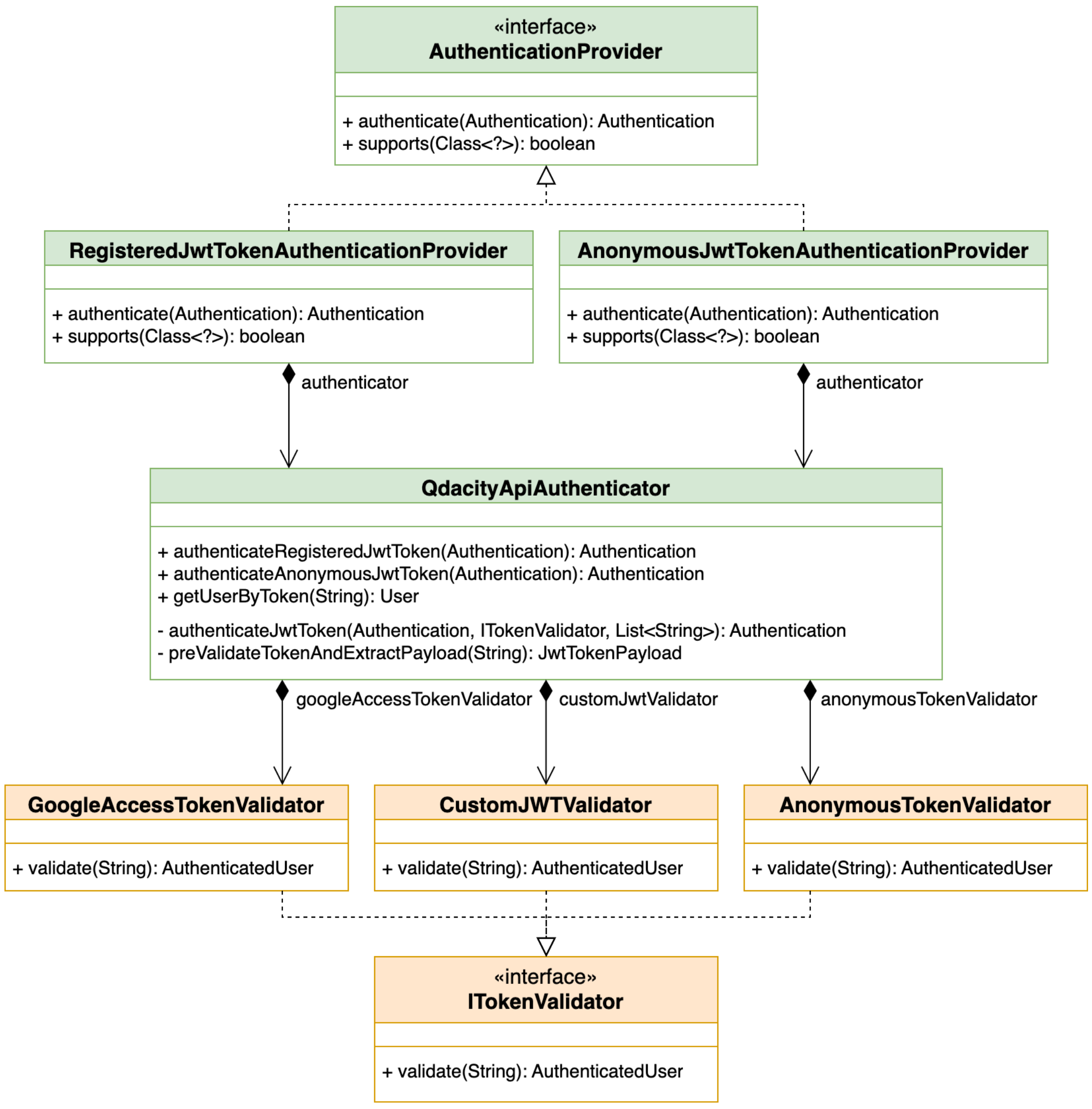

«interface»
AuthenticationProvider
+ authenticate(Authentication): Authentication
+ supports(Class<?>): boolean
RegisteredJwtTokenAuthenticationProvider
+ authenticate(Authentication): Authentication
+ supports(Class<?>): boolean
AnonymousJwtTokenAuthenticationProvider
+ authenticate(Authentication): Authentication
+ supports(Class<?>): boolean
authenticator
authenticator
QdacityApiAuthenticator
+ authenticateRegisteredJwtToken(Authentication): Authentication
+ authenticateAnonymousJwtToken(Authentication): Authentication
+ getUserByToken(String): User
- authenticateJwtToken(Authentication, ITokenValidator, List<String>): Authentication
- preValidateTokenAndExtractPayload(String): JwtTokenPayload
googleAccessTokenValidator
customJwtValidator
anonymousTokenValidator
GoogleAccessTokenValidator
+ validate(String): AuthenticatedUser
CustomJWTValidator
+ validate(String): AuthenticatedUser
AnonymousTokenValidator
+ validate(String): AuthenticatedUser
«interface»
ITokenValidator
+ validate(String): AuthenticatedUser

Figure 4.1: UML class diagram of the authentication mechanism

"typ", which specifies the token type and must equal "jwt". The payload holds two fields: "iss", the issuer, which must be "QDACity", and "authNetwork", which defines the authorization network. The authorization network could either be "EMAIL_PASSWORD" for registered users or "ANONYMOUS" for anonymous client communications. These fields are essential during the validation process handled by the token validators.

The Spring Security configuration in QDAcity defines multiple filter chains, each designed for different authentication scenarios. Five SecurityFilterChain instances are configured: one for paths that allow both registered and anonymous authentication, one for paths restricted to anonymous authentication only, one for

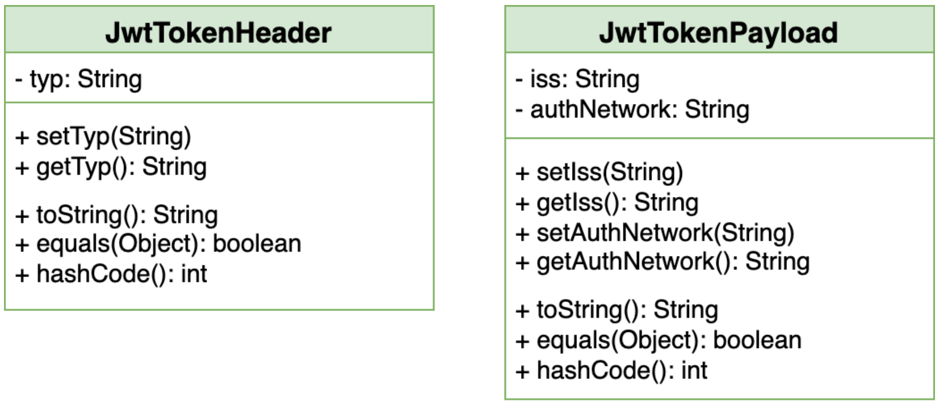


Figure 4.2: UML class diagram of the QDAcity JWT header and payload

API paths not requiring authentication, one for paths restricted to registered users only, and a default filter chain that permits all other requests. In the filter chain for registered and anonymous authentication, an OAuth 2.0 resource server is configured for handling Google JWT tokens. It automatically fetches the public verification keys from Google for token verification. This chain also registers the RegisteredJwtTokenAuthenticationProvider and AnonymousJwtTokenAuthenticationProvider classes to handle custom JWT tokens. The filter chain for anonymous authentication and the filter chain for registered authentication configure the respective authentication providers for their specific use cases. The default filter chain permits all requests, allowing unauthenticated access to paths not covered by the other chains.

Across all filter chains, some common configurations are applied. CSRF protection is disabled for performance reasons, as it is not necessary for a stateless REST API with JWT-based authentication.[2] Sessions are also disabled, ensuring the application remains stateless in line with RESTful principles. In summary, the QDAcity authentication process leverages a combination of custom JWT tokens, multiple authentication providers, and token validators integrated into Spring Security's filter chains to manage user authentication securely and efficiently. This system is designed to handle both registered and anonymous users, providing secure access to the API while maintaining flexibility for different authentication use cases.

As an example, the sequence diagram in figure 4.3 illustrates a positive authentication process for a registered user with a custom JWT token. The process starts at the ProviderManager of the filter chain, calling first the supports method of the RegisteredJwtTokenAuthenticationProvider and subsequently the authenticate method with the authentication object. The provider invokes the authenticateRegisteredJwtToken method of QdacityApiAuthenticator, which executes

[2]https://www.baeldung.com/spring-security-csrf

its central logic authenticateJwtToken with the CustomJWTValidator instance and the authorization networks "EMAIL_PASSWORD" and "GOOGLE". This method consequently parses the jwtTokenPayload via the preValidateTokenAndExtractPayload helper and selects the CustomJWTValidator, approving the token. After this process, the authenticateJwtToken method sets the authentication object to authenticated and returns it back to the RegisteredJwtTokenAuthenticationProvider. This provider also returns the authentication to the ProviderManager, which forwards the information to the above Spring Security classes as the AuthenticationManager and many more.

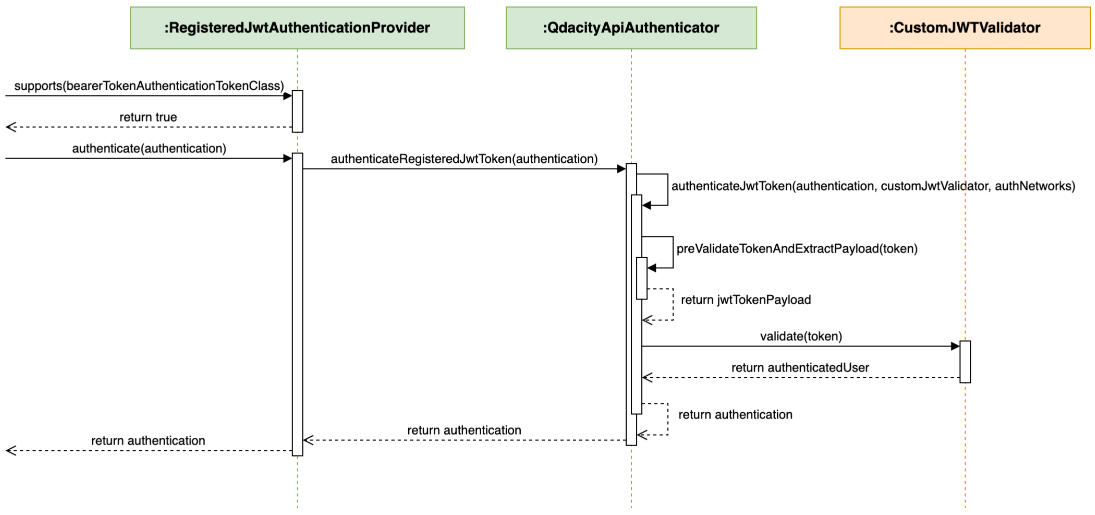

:RegisteredJwtAuthenticationProvider
:QdacityApiAuthenticator
:CustomJWTValidator
supports(bearerTokenAuthenticationTokenClass)
return true
authenticate(authentication)
authenticateRegisteredJwtToken(authentication)
authenticateJwtToken(authentication, customJwtValidator, authNetworks)
preValidateTokenAndExtractPayload(token)
return jwtTokenPayload
validate(token)
return authenticatedUser
return authentication
return authentication
return authentication

Figure 4.3: UML sequence diagram of the authentication mechanism

4.1.7 Endpoint Wrappers and Authorization

Besides the authentication, the Spring Boot backend requires controllers that define the API endpoints and their operations. This chapter explains only the design and the inner workings of such an exemplary QDAcity endpoint class called DocumentEndpoint, while the next subsection 4.1.8 gives some insights into the implementation of Spring Boot endpoint classes in the context of QDAcity.

Static View

As shown in figure 4.4, the DocumentEndpoint class plays a key role in the document insertion process by encapsulating the interaction between various system components responsible for access to user data, authorization, and the actual document insertion. It serves as the entry point for requests made by clients, exposing the insertDocument method, which is designed to handle user authorization checks and the eventual insertion of a document into the system.

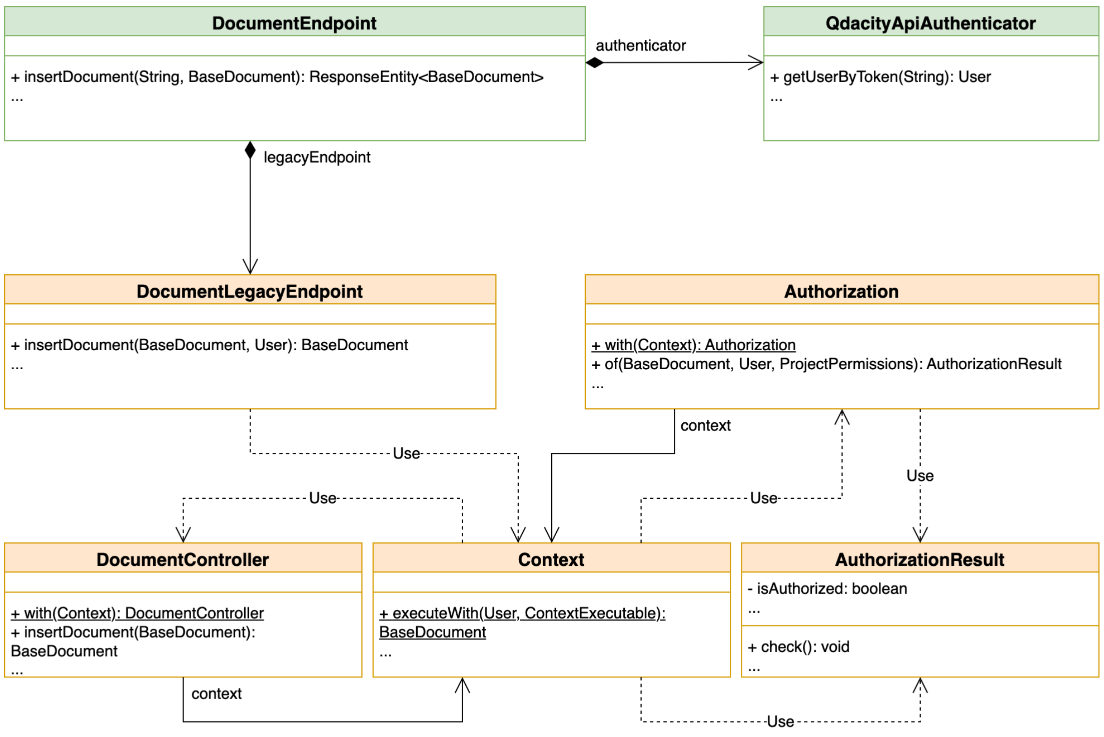

DocumentEndpoint
+ insertDocument(String, BaseDocument): ResponseEntity<BaseDocument>
...
authenticator
QdacityApiAuthenticator
+ getUserByToken(String): User
...
legacyEndpoint
DocumentLegacyEndpoint
+ insertDocument(BaseDocument, User): BaseDocument
...
Authorization
+ with(Context): Authorization
+ of(BaseDocument, User, ProjectPermissions): AuthorizationResult
...
context
Use
Use
Use
Use
DocumentController
+ with(Context): DocumentController
+ insertDocument(BaseDocument): BaseDocument
...
Context
+ executeWith(User, ContextExecutable): BaseDocument
...
AuthorizationResult
- isAuthorized: boolean
...
+ check(): void
...
context
Use

Figure 4.4: UML class diagram of the document endpoint as an example

This method accepts a token string, tokenHeader, which represents the authentication token of the user, and a BaseDocument object, which contains the data to be inserted. The DocumentEndpoint does not handle the complete process on its own but delegates key responsibilities to several collaborating components.

Upon receiving a request, the first step is loading the User object via the token from the database. This is achieved by calling the getUserByToken method of the QdacityApiAuthenticator class. This method validates the token and returns a user corresponding to the token provided. If the authentication fails, the request is terminated and an appropriate response is returned.

Once the user is authenticated, the responsibility of handling the actual document insertion is delegated to the DocumentLegacyEndpoint class. The DocumentEndpoint forwards the document and authenticated user to the insertDocument method in the DocumentLegacyEndpoint, which manages the insertion inside the legacy system. The relationship between the DocumentEndpoint and DocumentLegacyEndpoint classes follows the facade pattern (Gamma et al., 1994).

Facade Pattern

The DocumentEndpoint acts as a facade, simplifying the integration of the new backend framework Spring Boot. Clients interact solely with the DocumentEndpoint, which hides the complexity of the underlying legacy systems. This design is advantageous because it decouples the external interface from the internal implementation, promoting the migration process.

In the facade pattern, the DocumentEndpoint serves as a high-level interface, which consolidates operations related to document insertion. Rather than interacting directly with the legacy system, clients are shielded from the legacy endpoint classes. The actual document insertion is handled by the DocumentLegacyEndpoint and its dependencies.

The DocumentLegacyEndpoint manages the legacy operations, while the DocumentEndpoint ensures that data mapping, data validation, and loading of the user data are completed before delegating the document insertion process. By decoupling these responsibilities, the facade pattern promotes the separation of concerns during the migration process.

Runtime View

The sequence diagram in figure 1 of the appendices illustrates the flow of operations during the execution of the insertDocument method. Initially, the client invokes the insertDocument method on the DocumentEndpoint, passing in the token header and BaseDocument object. The DocumentEndpoint loads the user via the QdacityApiAuthenticator, which returns a User object if the token is valid. The DocumentEndpoint then forwards the document and the user to the DocumentLegacyEndpoint.

Within the DocumentLegacyEndpoint instance, the insertion logic is executed within a Context object. The context ensures that the operation is conducted in a controlled environment. The authorization process is carried out via the Authorization class, which verifies the user's permissions and returns an AuthorizationResult. If the user is authorized, the document is inserted into the system, and a ResponseEntity object is returned to the client as confirmation of a successful operation. If the user lacks the necessary permissions, the process terminates early, returning an error response to the client.

Design Evaluation

In summary, the DocumentEndpoint class revolves around providing a Spring Boot interface for document operations while delegating responsibilities such as accessing user data, authorization, and interaction with legacy systems to other components. By utilizing the facade pattern, the endpoint design supports the integration of Spring Boot while preventing code duplication during the migration process.

4.1.8 API Implementation and Specification

The API implementation and specification of QDAcity utilize a structured approach based on Spring Boot and OpenAPI, integrating various annotations and configurations. Below is a detailed description of these aspects, with a breakdown into key sections, including endpoint annotations, OpenAPI configuration, the generation and delivery of the OpenAPI specification, and the visualization of the API in the SwaggerUI.

Endpoint Annotations

In QDAcity, the API endpoints are implemented using Spring's annotation-based approach, making the endpoints easy to manage and document. At the class level, the @RestController annotation registers endpoint classes as REST controllers within the Spring context, enabling them to handle HTTP requests (see appendix A.2). The @RequestMapping annotation specifies the base path for an endpoint, determining the URL paths managed by the class. Lombok's @RequiredArgsConstructor simplifies dependency injection by automatically generating constructors for required fields.

The class-level annotations also include OpenAPI and Swagger documentation, such as @ApiResponses, which defines possible HTTP response codes like 200 (OK), 204 (no content), 400 (bad request), 404 (not found), and 500 (internal server error). Each response is further described using @ApiResponse, which includes the response code and a brief description. Additionally, the @Tag annotation groups the API operations of the endpoint class.

The methods in these classes are responsible for handling specific HTTP requests. They are annotated with mappings via @PutMapping, @PostMapping, @GetMapping, and @DeleteMapping to define the HTTP methods (i.e., PUT, POST, GET, DELETE) that the operations handle (see appendix A.3). These mappings, paired with additional path specifications, allow the API to support create, read, update, and delete (CRUD) operations or more complex business operations.

Moreover, the method level includes several annotations to specify its behavior. The @Operation annotation defines the OpenAPI documentation for the method, providing an operation ID, a summary containing the group tag and the operation ID for searching the operation, and a detailed description. It also includes a @SecurityRequirement annotation to enforce JWT bearer authentication. Parameter annotations such as @RequestParam, @RequestHeader, and @RequestBody are used to define the expected input, whether they come from the URL query, the headers, or the request body. Finally, the method returns a ResponseEntity object, which wraps the HTTP status code and body, ensuring a well-formed response to the client.

OpenAPI Configuration

The OpenAPI configuration for QDAcity is managed in a class called OpenAPI30-Config, which registers the OpenAPI version 3.0 configuration within the Spring context. This class is annotated with @Configuration, making it available for the ApplicationContext. The @OpenAPIDefinition annotation provides general information about the API, including its title, version, and other properties that appear in the OpenAPI specification.

The configuration also includes security settings, which are defined using the @SecurityScheme annotation. This annotation describes the API's security requirements, particularly the use of JWT bearer authentication for securing endpoints. Additionally, @Bean methods are used to create instances of GroupedOpenApi, which group the API endpoints in the SwaggerUI. In this case, endpoint operations are grouped by their endpoint base path to appear in the right SwaggerUI sub-page.

OpenAPI Specification Generation and Delivery

The OpenAPI specification for the API is generated automatically and packaged as a static file with the QDAcity-API module. This process is tightly integrated with the Maven build system.

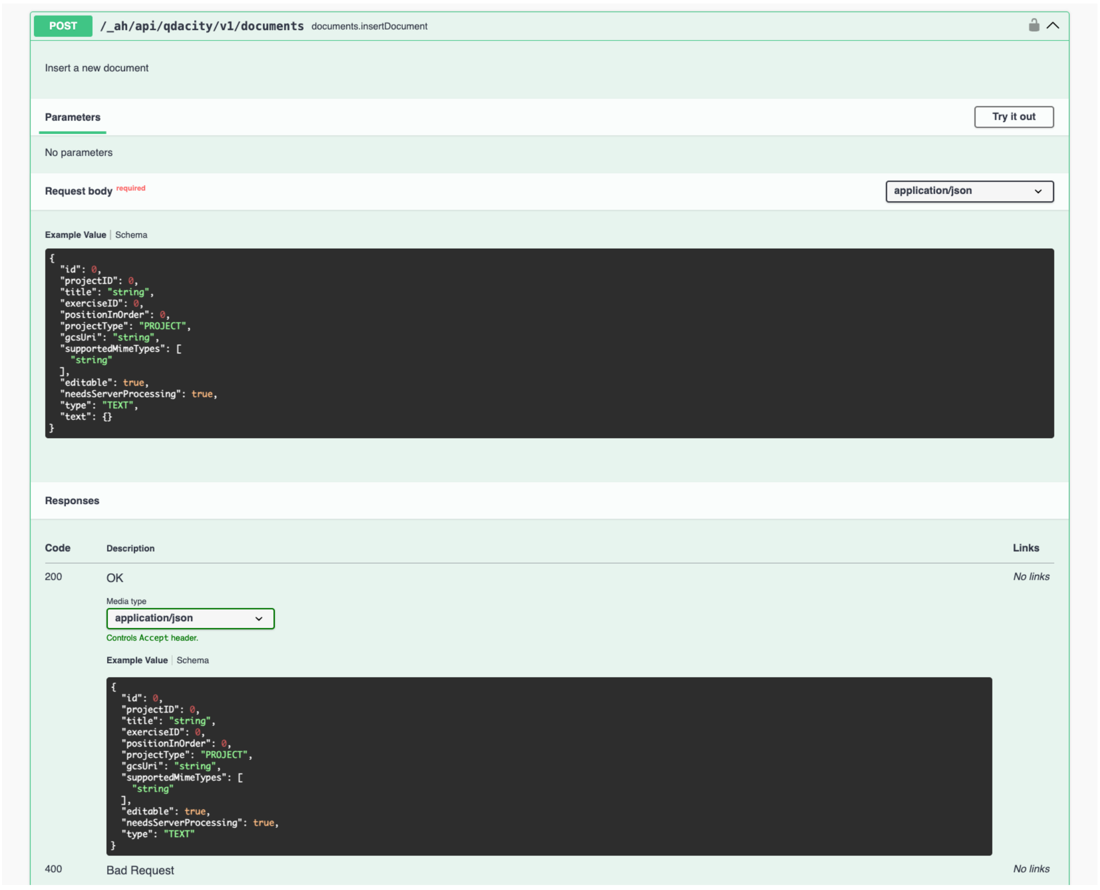
During the Maven build process, the OpenAPI specification (openapi-vX.json) is automatically generated at server startup through the SpringDoc-OpenAPI Maven plugin. The specification is hosted under the API docs endpoints, making it available for API clients. However, to package the OpenAPI specification as a file into the WAR, additional steps are taken. The generated openapi-vX.json is downloaded during the integration-test phase and stored in the WEB-INF directory of the QDAcity-API module. The Maven WAR plugin then repackages the WAR file, including the OpenAPI specification in its root directory.

The build process also incorporates platform-specific handling for managing server processes. On Linux systems, the appengine-stop execution is used to stop the server, while on MacOS and Windows, platform-specific scripts are executed to kill the running process, as the default appengine-stop execution does not terminate the server properly on these platforms.

The automated generation of the OpenAPI specification is a remnant of earlier migration attempts, during which the openapi-vX.json was served as a static file via the Jetty web server. However, the OpenAPI specification is now provided through an API docs endpoint at runtime, meaning it no longer needs to be stored as a file. Nevertheless, packaging the OpenAPI specification has the advantage that each version of the QDAcity-API WAR includes a corresponding specification, ensuring that it is also released along with the application.

Representation in SwaggerUI

Once the OpenAPI specification is generated, it is displayed in SwaggerUI, providing a clear, interactive interface for developers to explore and test the API. Endpoint operations are grouped by class and each operation method is documented with its corresponding path, HTTP method, parameters, response codes, and security requirements (see figure 4.5). The grouping of endpoint operations, combined with detailed annotations for each operation, ensures a well-documented and easily navigable API within SwaggerUI. This allows developers to quickly identify the available operations and their associated input parameters and output structures while also adhering to security protocols like bearer authentication.



`/_ah/api/qdacity/v1/documents` documents.insertDocument

Insert a new document

Parameters

No parameters

Request body required

Example Value | Schema

```
{
  "id": 0,
  "projectID": 0,
  "title": "string",
  "exerciseID": 0,
  "positionInOrder": 0,
  "projectType": "PROJECT",
  "gcsUri": "string",
  "supportedMimeTypes": [
    "string"
  ],
  "editable": true,
  "needsServerProcessing": true,
  "type": "TEXT",
  "text": {}
}
```

Responses

Code	Description	Links
200	OK	No links
400	Bad Request	No links

Media type: application/json

Controls Accept header.

Example Value | Schema

```
{
  "id": 0,
  "projectID": 0,
  "title": "string",
  "exerciseID": 0,
  "positionInOrder": 0,
  "projectType": "PROJECT",
  "gcsUri": "string",
  "supportedMimeTypes": [
    "string"
  ],
  "editable": true,
  "needsServerProcessing": true,
  "type": "TEXT"
}
```

Figure 4.5: QDAcity SwaggerUI operation window of insertDocument as an example for manually testing and documenting the QDAcity API operations

4.2 Frontend

The design and implementation chapter of the QDAcity frontend covers the components that were affected or introduced by the backend framework migration.

It especially details the handling and integration of Swagger Client and its responses and errors. Like the backend before, the frontend chapter starts with the setup of Swagger Client (see chapter 4.2.1), while the following sub-chapters explain the client handling (see chapter 4.2.2), the client execution (see chapter 4.2.3), and the endpoint services (see chapter 4.2.4).

4.2.1 Swagger Client Setup

The Swagger Client dependency is installed via NPM and therefore part of the package.json and package-lock.json files of the QDAcity frontend. The client does not require any specific configurations or setup scripts but is fully usable right after the installation.

4.2.2 Client Handling by QdacityApiClient

The QdacityApiClient class of the QDAcity frontend serves as the central client for managing communication between the frontend and backend. It has three main fields: "token", "swaggerClient", and "apis". The token field is initialized by the frontend's authentication mechanisms, specifically the AnonymousAuth and AuthenticationProvider classes, as well as the service worker. The "swaggerClient" field is a promise that is instantiated using the API docs endpoint of the backend. This SwaggerClient instance fetches and caches the latest version of the OpenAPI specification, ensuring that the specification is re-fetched whenever the version changes. The client also includes a request interceptor, which sets the authorization header with the bearer JWT token for every request, and a response interceptor, which maps the response body to a format compatible with the legacy GAPI Client or the legacy API, thereby aligning the migrated API with the existing frontend logic.

The "apis" field is initialized as a function that accesses the SwaggerClient instance, returning a promise that resolves into a combined object, merging the OpenAPI specification details with method callbacks via the SwaggerClient's tag interface. This tag interface allows calling API operations as JavaScript methods via their endpoint tag and operation ID. The constructor initializes the "token" field as null.

Several asynchronous "getter" methods are available for retrieving specific information about a given API operation. These methods take the endpoint tag and operation ID as parameters and access the relevant operation from the "apis" field.

The class also includes a method getRequestData that converts a request into an object with the fields "id", "params", and "body", according to the OpenAPI specification of the SwaggerClient instance. Additionally, the class provides a method called "then", which executes a given function through the promise stored in the "apis" field, ensuring asynchronous functionality.

Lastly, the constant qdacityApiClient holds an instance of the QdacityApiClient class and serves as a central client for all endpoint service classes and authentication logic throughout the application.

4.2.3 Client Execution by Promisizer

The Promisizer class performs requests via the QdacityApiClient instance and consequently handles the responses or errors. Therefore, the activity diagram in figure 4.6 visualizes this handling logic on a more abstract level because the actual JavaScript code handles every activity via Promises in an asynchronous fashion. This would be very difficult to illustrate with a more formal diagram type like a UML sequence diagram. Thus, the following paragraphs describe this process of figure 4.6.

Performing Requests

The execution of an API call in the Promisizer class begins by invoking the API method callback with a set of parameters and options. This triggers the API method callback to execute, returning a result.

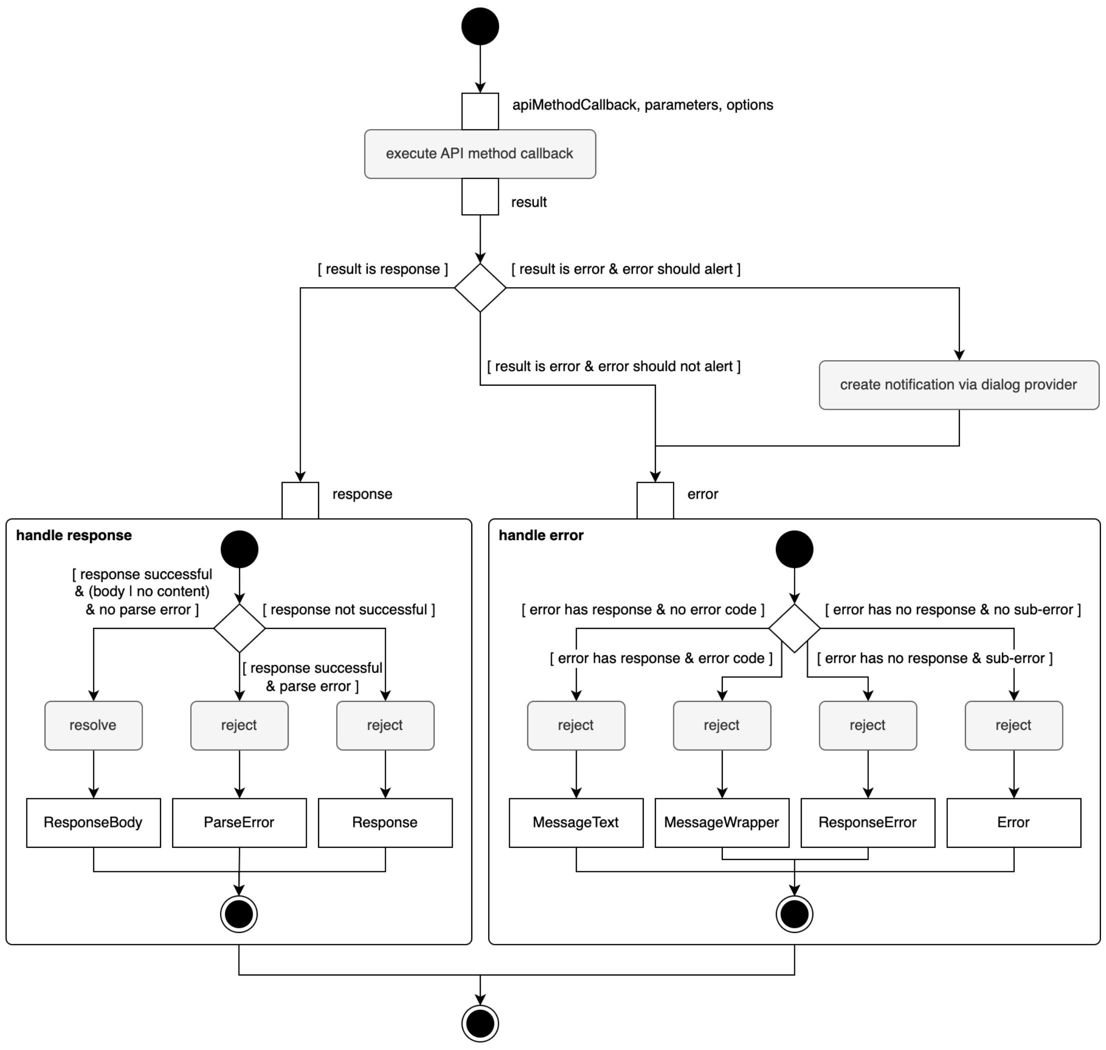
If the result is a valid response, the process proceeds to handle the response. Otherwise, if the result is an error, two scenarios are possible. First, if the error requires an alert, a notification is generated via the dialog provider. Second, if the error does not require an alert, the error will be handled without a notification.

Handling Responses

Once a valid response is obtained, three cases are considered. If the response is successful, a response body or a 204 (no content) status code is present, and no parse error occurs, the promise is resolved and the parsed response body is returned. If the response is not successful, the promise is rejected and the response object is returned. If a response body exists but a parse error occurs, the parse error object is returned and the promise is rejected.

Handling Errors

If the result is an error, the error is handled based on specific conditions. First, if the error has a response but no error code, the promise is rejected with a message text. Second, if the error has a response and an error code, the promise is rejected with a message wrapper object. Third, if the error has no response but contains a sub-error, the promise is rejected with a response error object. Finally, if the error has neither a response nor a sub-error, a generic error is thrown and the promise is rejected.



apiMethodCallback, parameters, options
execute API method callback
result
[result is response]
[result is error & error should alert]
[result is error & error should not alert]
create notification via dialog provider
response
error
handle response
[response successful & (body | no content) & no parse error]
[response not successful]
[response successful & parse error]
resolve
reject
reject
ResponseBody
ParseError
Response
handle error
[error has response & no error code]
[error has no response & no sub-error]
[error has response & error code]
[error has no response & sub-error]
reject
reject
reject
reject
MessageText
MessageWrapper
ResponseError
Error

Figure 4.6: UML activity diagram of the Promisizer logic

In either case – whether a response is successfully handled or an error occurs – the process is completed by resolving or rejecting the promise accordingly.

4.2.4 Endpoint Services

The endpoint services provide methods for accessing the backend API of QDAcity through the use of the QdacityApiClient and the Promisizer class. Two methods of the DocumentsEndpoint class serve as an example of how this interaction is implemented (see appendix B).

The getDocuments method is a static function that retrieves documents from the backend. It takes two parameters, a project ID and a project type. The method constructs an API method callback, which executes the API operation via the SwaggerClient instance inside the QdacityApiClient object. This callback is passed to the makeResponseHandlerPromise function of the Promisizer, along with the operation-specific parameters as a second argument. These parameters contain HTTP path parameters, query parameters, and headers. In this specific case, the method finally chains a last "then" call that processes the response, extracting and returning the items from the response body object.

The insertDocument method, also a static function, allows for inserting a new document. It takes a single parameter, doc, which represents the document to be inserted. Similar to the getDocuments method, an API method callback is created. The Promisizer function makeResponseHandlerPromise is called with this callback and an empty object for the parameters, while the document to be inserted is passed as part of the request body option in the third argument. The method does not include a "then" call, indicating that it returns a promise that will be resolved or rejected based on the response from the Promisizer.

In both methods, the QdacityApiClient ensures that the appropriate API methods are called, while the Promisizer handles the promise-based workflow, ensuring that the response or error is properly processed and returned to the calling context.

5 Evaluation

The evaluation chapter assesses how well the proposed solutions meet the specified functional and non-functional requirements defined in chapter 2. By analyzing the architecture, design, and implementation, this chapter aims to determine whether the system fulfills the intended functionality (see chapter 5.1), while adhering to performance, compatibility, security, and other non-functional constraints (see chapter 5.2). Through this evaluation, the strengths and weaknesses of the implemented solutions are highlighted, providing insight into their overall effectiveness and identifying areas for potential improvements, which are covered in chapter 6.

5.1 Functional Requirements

All in all, the presented solution of the framework migration through architecture, design, and implementation fits all functional requirements which are defined in chapter 2.1. Especially the fulfillment of the overarching FRQ-1 ensures that the application features are preserved and the users are not affected by the migration. However, this was to be expected, as this thesis is carrying out a technology migration and is therefore less feature-heavy. The focus is more on the non-functional requirements, evaluated in chapter 5.2.

5.1.1 API Specification

Table 5.1 evaluates the functional requirements related to API responses against the solutions presented in the given chapters. As already mentioned, the OpenAPI specification of QDAcity fulfills all defined requirements of chapter 2.1.1.

The initial requirements pertain to the definition of endpoints and operations and, like the other API specification requirements, are covered in chapter 4.1.8. In contrast, the requirements related to potential error handling are supplemented by additional chapters, which are also referenced in the table (see FRQ-13 to FRQ-18). This approach of referencing the chapters addressing the mentioned requirements is applied to the other sub-chapters of the evaluation chapter as well.

FRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
2-12			X	4.1.7, 4.1.8
13			X	4.1.4, 4.1.8
14-15			X	4.1.4, 4.1.6, 4.1.8
16			X	4.1.8
17-18			X	4.1.4, 4.1.8

Table 5.1: Evaluation of the functional requirements related to API responses

5.1.2 API Security

Table 5.2 evaluates the functional requirements related to API security against the solutions presented in the given chapters. The access control of QDAcity and the Spring Security mechanisms fulfill all defined requirements of chapter 2.1.2.

FRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
19-26			X	4.1.6, 4.1.7

Table 5.2: Evaluation of the functional requirements related to the API security

5.1.3 API Client for Manual Testing

Table 5.3 evaluates the functional requirements related to the API client for testing against the solutions presented in the given chapters. The SwaggerUI of QDAcity fulfills all of the requirements defined in chapter 2.1.3.

FRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
27-34			X	3.2.7, 3.6.4, 4.1.8

Table 5.3: Evaluation of the functional requirements related to the API client for manual testing

5.2 Non-Functional Requirements

In addition to the functional requirements, the architecture, design, and implementation of the migrated solution meet the non-functional requirements in most cases (see chapter 2.2). Especially, the analysis of the performance requirements, a sub-category of the quality requirements, reveals potential for further performance optimization, while for example the technological requirements are fulfilled completely.

5.2.1 Technological Requirements

Table 5.4 evaluates the technological requirements related to the environment and interfaces against the solutions presented in the given chapters. The technology stack and the API design of QDAcity fulfill the requirements TRQ-1 to TRQ-5 defined in chapter 2.2.1.

Furthermore, SwaggerUI helps ensure that the QDAcity API will fulfill the Open-API 3.0 standard (i.e., TRQ-4) in the future by ignoring non-compliant parameters or operations in the user interface, when developers use it to check new endpoints and endpoint operations.

TRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
1-3			X	3.1.3
4			X	4.1.8
5			X	4.1.6

Table 5.4: Evaluation of the technological requirements

5.2.2 Quality Requirements

This chapter evaluates the quality requirements (see chapter 2.2.2) related to performance, compatibility, usability, reliability, security, maintainability, and portability (i.e., the seven "true" quality dimensions) against the solutions presented in the referenced chapters. The quality of the migrated QDAcity backend is on a very good level in general but requires a detailed evaluation in the next paragraphs.

Performance

Table 5.5 evaluates the quality requirements related to performance against the solutions presented in the given chapters. The caching mechanism of the Swagger Client fulfills the performance requirement QRQ-1. To make reliable statements about the quantitative performance of a system, a load and performance test would have to be carried out. However, this is beyond the scope of this thesis. In chapter 3.5.2, there is already a short mention of quantitative results for a basic setup with Spring Boot and Jetty, but this does not match the scope and environment of QDAcity.

Therefore, the quantitatively measurable requirements are evaluated via an observability tool of Google App Engine, called metrics explorer (see figure 2 in the appendices). The used metrics are the CPU utilization of QRQ-2, the memory usage of QRQ-3, and the response latency of QRQ-4, compared between QDAcity version 230 with Google Endpoints and versions 231 to 236 with Spring Boot.

QRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
1			X	4.2.2
2-4		X		–

Table 5.5: Evaluation of the quality requirements related to performance

In addition, the response count weights the observations and conclusions drawn from the other charts in appendix C. Each of these line charts, which were taken from the metrics explorer, visualizes two different aggregations of the underlying measuring points. The 99th percentile is the measure greater than 99 percent of the series, while the mean is a simple average per time interval. Therefore, these aggregations together pay attention to outliers and extreme values with the 99th percentile, while the mean provides a better understanding of the system performance in general.

A comparison of the mean values shows that the 10 percent mark for CPU utilization (see appendix C.1) is still rarely exceeded in general. However, the 99th percentile indicates that CPU utilization before the migration to Spring Boot was very stable and hardly deviated from the mean, whereas now there are frequent extreme spikes towards the 10 or even 20 percent CPU utilization. While this value is not yet alarming, it does represent a disadvantage compared to the solution with Google Endpoints.

Similar insights can be drawn from memory usage (see appendix C.2), which, both before and after, mostly ranged between 400 and 700 MiB. However, directly after the first Spring Boot release with version 231 (on the 19th of September 2024) and correlated with CPU utilization, memory usage spikes sharply, exceeding the upper limit of available memory in production (slightly above 750 MiB). This can, in turn, lead to repeated server instance crashes, even though these are not directly reflected by the occasional dips in the mean. These dips occurred due to deployments of new app versions from 231 to 236. However, after multiple performance optimizations, the memory usage for later versions with Spring Boot stabilizes near the upper limit of around 750 MiB.

The response count metric (see appendix C.3) also shows that the extreme increases in memory usage and CPU utilization occur particularly during phases of high load caused by many HTTP requests and responses. This phenomenon applies to version 230 with Google Endpoints as well as the newer versions with Spring Boot.

Furthermore, the response latency (see appendix C.4) best illustrates the performance issues and potentials of the Spring Boot versions at the same time. While version 230 without Spring Boot only experienced extreme spikes after the start of a new server instance and then stabilized, these extreme spikes occur more frequently and unpredictably with Spring Boot versions. In some cases, response latency increases after starting a new server instance, in others, due to a higher request load. Nevertheless, the most extreme values have been reduced noticeably (mostly up to 1 minute) compared to version 230 with the Google Endpoints Framework (mostly up to 3-4 minutes). Additionally, a separate examination of the mean shows that the response latency generally stays below 1 to 2 seconds. Therefore, it's clear that response latency has not been stable and has become even more unstable and unpredictable due to the migration.

The target would have been a stable response latency of under 1 second, or at most 2 seconds, to avoid noticeable slowdowns of the app. Memory usage also leaves much to be desired but seems to stabilize (stable 600-750 MiB >= 600 MiB), while the CPU utilization has remained within an acceptable range but exceeds the expected value by far (most peeks in 20-40% >= 20%). As a result, the overall quantitative performance requirements have not been met completely, except in average or best-case scenarios.

Compatibility

Table 5.6 evaluates the quality requirements related to compatibility against the solutions presented in the given chapters. The migrated endpoints based on the facade pattern (Gamma et al., 1994) fulfill the requirements QRQ-5 to QRQ-7 as far as possible and useful, while they ensure QRQ-8 to QRQ-11 completely. In addition, the API client in the frontend handles some compatibility issues of the migrated QDAcity API compared to the legacy API, which should be refactored in the future as described in chapter 6.1.4. Furthermore, the combined access control via Spring Security and the legacy Authorization class fulfill QRQ-12.

QRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
5-11			X	4.1.8
12			X	4.1.6, 4.1.7

Table 5.6: Evaluation of the quality requirements related to compatibility

Usability

Table 5.7 evaluates the quality requirements related to usability against the solutions presented in the given chapters. SwaggerUI stores a submitted ID token via the local application storage in the browser and thus persists them across subpages and page reloads. In the current configuration, it can not acquire tokens automatically, but it provides plugins and custom configurations for standardized logins like Google OAuth 2.0. Consequently, the QDAcity SwaggerUI does fulfill the usability requirement QRQ-13 partially.

QRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
13		X		3.2.7

Table 5.7: Evaluation of the quality requirements related to usability

Reliability

Table 5.8 evaluates the quality requirements related to reliability against the solutions presented in the given chapters. The adapted integration and deployment process of QDAcity ensures QRQ-14 and the API versioning in the OpenAPI specification fulfills QRQ-15.

QRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
14			X	–
15			X	4.1.8

Table 5.8: Evaluation of the quality requirements related to reliability

Security

Table 5.9 evaluates the quality requirements related to security against the solutions presented in the given chapters. The appengine-web.xml of the migrated backend enables TLS encryption on Google App Engine, thus fulfilling the security requirement QRQ-16. Additionally, during the migration of the endpoint classes and their operations, care was taken to ensure that sensitive data is only sent in the request body, which also meets QRQ-17. Furthermore, the facade pattern (Gamma et al., 1994) of the endpoints ensures that the Authorization class in the legacy endpoint classes continues to handle user permission checks (see QRQ-18). The debugging level is currently not configured separately for QDAcity. It is always "info" and therefore fulfills QRQ-19, but misses the QRQ-20 requirement. During the setup of the Spring Boot backend, an analysis of vulnerable dependencies was conducted and appropriate actions (e.g., upgrading a transitive dependency) were taken (see QRQ-21).

QRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
16-17			X	–
18			X	4.1.7
19			X	–
20	X			–
21			X	3.1.3

Table 5.9: Evaluation of the quality requirements related to security

Since QDAcity does not yet use automated penetration testing, no further statements can be made about the "actual" security of the backend.

Maintainability

Table 5.10 evaluates the quality requirements related to maintainability against the solutions presented in the given chapters. The existing GitLab CI for QDAcity meets the QRQ-22 requirement and also fulfills the minimal branch coverage requirement from QRQ-23 through the integrated test jobs. The general documentation for the QDAcity product and project can still be found in the GitLab Wiki, and it was updated in several places during and after the backend migration (see QRQ-24 and QRQ-25). SwaggerUI, as frequently mentioned, serves as both an API client for manual testing and as API documentation (see QRQ-26). Additionally, this thesis documents the migration efforts and thus fulfills QRQ-27. Of course, at the time of this master thesis, there is still no experience with the development and operation of the migrated backend, which makes it difficult to make statements about future maintainability.

QRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
22-23			X	3.4.3
24-25			X	–
26			X	3.2.7
27			X	–

Table 5.10: Evaluation of the quality requirements related to maintainability

Portability

Table 5.11 evaluates the quality requirements related to portability against the solutions presented in the given chapters. The chapter 6.1.2 describes, that the app setup, integration, and deployment can be adapted to native Google App Engine Gen 2 without the bundled Gen 1 services (see QRQ-28 to QRQ-30).

QRQ(s)	Missed	Incomplete	Fulfilled	Addressed in chapter(s)
28-30			X	3.4.2, 4.1.2, 6.1.2

Table 5.11: Evaluation of the quality requirements related to portability

6 Outlook

The outlook chapter identifies areas for future work (see chapter 6.1), highlighting tasks that remain unresolved or require further development. Additionally, it provides recommendations to guide future modifications of QDAcity (see chapter 6.2) to improve the system design, the tests, and the implementations of certain components based on further technologies.

6.1 Future Work

The completion of this thesis marks significant progress in the migration of QDAcity's backend framework. However, several areas require further exploration and development beyond the scope of this thesis. These future tasks involve addressing technical challenges, optimizing performance, and expanding the solutions. In particular, additional work is needed to refine integration processes, improve the efficiency of certain components, and ensure the long-term maintainability of the system. This chapter outlines the key areas for future development that will enhance the migration and its impact on QDAcity's backend architecture.

6.1.1 Backend Build Optimization

The future work on backend build optimization for QDAcity involves several key tasks aimed at simplifying and modernizing the build process. First, the core QDAcity module needs to transition from WAR to JAR packaging because it is now a dependency of the QDAcity-API module. As part of this switch, several outdated plugins, such as the Maven WAR plugin, Maven JAR plugin, App Engine Maven plugin, and Endpoints Framework Maven plugin, should be removed, as they will no longer be necessary for the new build configuration. Additionally, the QDAcity-API module needs to be adjusted to correctly import the JAR file from the core module without using a "dependency" classifier, which is no longer required. Legacy configuration files like appengine-web.xml and web.xml, as well as other obsolete files, should be removed to clean up the project structure.

Further optimization involves eliminating environment interpolations and Gulp scripts that are tied to the core QDAcity module. In the long run, it may be possible to fully remove Gulp scripts from the project, further simplifying the build process and reducing maintenance overhead.

Another optional optimization would be to remove the generation process that bundles the OpenAPI file, as it is no longer used for initializing the SwaggerClient instance in the frontend (see chapter 4.1.8).

6.1.2 Embedded Web Server

To switch the Spring Boot application deployment from a provided to an embedded Jetty server, several key changes need to be made to the QDAcity-API module. Currently, Jetty is provided by the runtime environment in App Engine Gen 2 with bundled Gen 1 services (see chapter 4.1.2), but with native Gen 2 in the future, the application needs to handle its server lifecycle by embedding Jetty directly within the application. This also is the preferred approach of Spring Boot.

The first step in this transition is to modify the Jetty dependency in the pom.xml file. The current setup marks Jetty as a provided dependency, meaning it is expected to be supplied by the runtime environment. For an embedded setup, this "provided" scope must be removed, ensuring that Jetty is packaged and run as part of the application itself. This will allow the Spring Boot application to include the Jetty server as a fully integrated part of its runtime.

Another important change is switching the packaging format from WAR to JAR. The current deployment relies on a WAR file, which requires an external server like Jetty or Tomcat to execute. By switching to a JAR file, the application can bundle the embedded Jetty server, making it fully self-contained and executable. This requires modifying the pom.xml to replace the Maven WAR plugin with the JAR plugin, ensuring the application is built as an executable JAR.

Additionally, since the WAR-based deployment uses a ServletInitializer class for servlet configuration, it can be removed. In a JAR-based setup with an embedded server, Spring Boot manages the server and servlet lifecycle internally, making the ServletInitializer unnecessary.

The application's API base path also needs to be updated. In the current configuration, servlet paths combine the base path from the application.yml with their sub-paths. With the switch to an embedded Jetty server, these paths need to be adjusted to only the sub-paths, while ensuring that the servlet context path is correctly defined in the application.yml.

Lastly, the migration requires enabling Spring Boot's programmatic registration of servlets, filters, and listeners, which is typical in embedded server configurations. This includes removing any XML-based registrations in the web.xml, which are traditionally used for servlet registration in external server environments. Instead, these components should be registered directly through Spring Boot's annotations and Java-based configuration.

6.1.3 Endpoint Wrappers Upgrade

As described in chapter 4.1.7, the legacy endpoint classes were wrapped with endpoint classes using the facade pattern (Gamma et al., 1994). This was done to host two QDAcity APIs, one using the Google Endpoints framework and the other with Spring Boot. Following a successful migration to Spring Boot and a single hosted API, these Spring-based wrappers can fully replace the legacy endpoints. This requires copying the code from each legacy endpoint class into its corresponding wrapper and then deleting the legacy classes. An incremental refactoring approach would be advisable here, processing the individual endpoints in alphabetical order.

6.1.4 API Adapters Excision

The framework migration in the backend and the switch from GAPI Client to Swagger Client caused many compatibility issues between the frontend and backend. As a result, it was necessary to implement some adapter solutions in the QDAcity app on both sides of the HTTP-based API (see chapters 4.1.5 and 4.2.2).

On the backend side, the serialization of Blob and Long values was explicitly adjusted to simulate the behavior of the Google Endpoints framework. However, these specific adjustments to the Jackson ObjectMapper are not particularly intuitive and could lead to unwanted issues in the future. Therefore, at the very least, the serialization of Long values should be changed from strings to numeric values.

On the frontend side, a response interceptor was written in the QdacityApiClient class, which intercepts all API responses and adjusts their data structures to be compatible with the legacy backend. This allows the subsequent frontend logic to work correctly with the data. This adapter solution should be removed in the future by gradually refactoring the frontend logic to eliminate the need for these special cases.

Once all refactorings in the frontend and backend are completed, the QDAcity API will be used on both sides as originally specified. Data will remain in their intended data types and the data structures in the communication payload will remain consistent throughout the entire application.

6.1.5 Backend Modularization

After the successful migration of the backend framework, the current module structure of the QDAcity application needs to be reconsidered. The existing setup consists of a core QDAcity module and the QDAcity-API module, both of which are bundled under a parent module named QDAcity-Backend (see chapters 3.6.1 and 4.1.1). However, this structure is not aligned with the actual architecture of the system. Instead, it was originally designed as a compromise to resolve dependency conflicts, which are no longer relevant after the migration.

A more logical and maintainable modularization can be implemented by restructuring the backend into distinct layers that reflect the system architecture. The proposed structure would begin with a QDAcity-Parent parent module, replacing the existing QDAcity-Backend parent.

The first sub-module, QDAcity, would serve as the deployable application artifact and bundle other sub-modules like QDAcity-Endpoint, QDAcity-Business, QDAcity-Data, and potentially also QDAcity-Test-Core. It takes on the role previously handled by QDAcity-API, acting as the central point for deploying the entire application.

The endpoint layer, including all endpoint classes, would be housed in a new QDAcity-Endpoint sub-module. This module encapsulates all API endpoint logic, separating it from the business and data layers to improve maintainability and clarity.

The business layer, including all controller classes, would be contained within the QDAcity-Business sub-module. This module would manage the core business functionality and logic of the application.

The QDAcity-Data sub-module would focus on the data access layer, including all DAO classes. It would handle interactions with the database and other data sources, ensuring that data management is isolated from the business and endpoint layers.

To enforce clean separation and modular interaction between layers, interface contracts would be introduced. The QDAcity-Business-Contract sub-module would contain the interfaces for the business controllers, which would be imported by both the QDAcity-Endpoint and QDAcity-Business modules. This would serve as a contract between the endpoint layer and the business layer, ensuring that the implementation details remain encapsulated.

Similarly, the QDAcity-Data-Contract sub-module would define the interfaces for the DAOs, which would be imported by both the QDAcity-Business and QDAcity-Data modules. This contract would enforce separation between the business and data access layers.

Another QDAcity-Data-Core sub-module would contain the data models used in the entire application. Therefore, all other sub-modules should import this module to get these data models.

Finally, testing would be streamlined by introducing a QDAcity-Test-Core sub-module. This module would provide utility classes and setup helpers for testing, which could be reused across multiple sub-modules. Unit tests would reside within the respective sub-modules, while integration and system tests would be grouped within the main QDAcity module, ensuring that the tests are organized according to the areas of functionality they cover.

This proposed modularization would result in a more cohesive backend architecture with clear boundaries between layers and reduced inter-module dependencies. These reduced dependencies enable Maven to build multiple backend modules in parallel, which accelerates the integration process in the pipeline and in the local environment.

6.2 Recommendations

Building on the foundation laid in the thesis and the future work before, several recommendations can be made to improve the solutions developed throughout this project. These improvements focus on enhancing the API tests, optimizing system performance, and adopting best practices to ensure sustainability. The recommendations in this chapter provide actionable suggestions for refining the existing solutions, addressing potential weaknesses, and exploring new technologies or methodologies that could further streamline the backend framework and its integration with other system components.

6.2.1 Endpoint Tests via Spring MockMvc

Currently, the endpoint classes in the QDAcity backend are only tested through simple method calls. This approach limits the scope of testing, as it focuses solely on individual method logic without considering the full behavior of the application in a real web environment. To improve the depth and accuracy of testing, switching to Spring MockMvc[1] is recommended.

Spring MockMvc is a comprehensive testing framework provided by the Spring Boot Starter Test dependency that allows developers to test various parts of the application, including web layers, in an environment closely resembling production. It includes support for loading the Spring application context, testing the

[1]https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html

interaction between components, and ensuring the correct behavior of the entire system.

Spring MockMvc is a tool specifically designed to test Spring MVC controllers, that are in the context of QDAcity called endpoints. It allows testing the behavior of web requests without actually starting a full web server. With MockMvc, tests can simulate HTTP requests and assert the correctness of responses, such as checking status codes, response bodies, headers, and more.

Using Spring MockMvc provides significant advantages over simple method calls. Instead of just testing isolated logic, it allows for testing the full interaction between the controller and other parts of the application, such as services and data layers. This means that issues related to request handling, serialization, and response formatting can be caught early. Additionally, MockMvc can validate if the controller behaves correctly under different HTTP methods (GET, POST, PUT, DELETE), improving the reliability of the API as a whole.

By moving from simple method-level testing to Spring Tests via MockMvc, the non-measured test coverage will increase, providing more confidence that the application behaves as expected in real-world scenarios. This change ensures that endpoint behavior is completely validated in an environment that simulates how the API will function in production.

6.2.2 Authorization via Spring Security

In the QDAcity backend, authentication is handled in the filter chain (see chapter 4.1.6), while authorization is performed within the individual endpoint methods (see chapter 4.1.7). This separation of concerns can lead to scattered and inconsistent security logic throughout the codebase. Furthermore, it is highly inefficient as it must authenticate twice to get the user for the authorization logic. To simplify and centralize security management, it is recommended to switch to using Spring Security.

Spring Security allows both authentication[2] and authorization[3] to be managed centrally, either entirely within the filter chain[4] or by using annotations like @PreAuthorize on methods[5]. This approach ensures that both authentication (verifying the user's identity) and authorization (checking user permissions) are handled before requests reach the endpoint layer. With Spring Security, security logic is applied consistently, reducing duplication and improving maintainability.

[2]https://docs.spring.io/spring-security/reference/servlet/authentication/index.html

[3]https://docs.spring.io/spring-security/reference/servlet/authorization/index.html

[4]https://docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html

[5]https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html

In this new setup, authentication will be performed in the filter chain by extracting and verifying tokens from the HTTP request, such as JWTs. After the token is validated, the user's permissions and roles are loaded from a repository (typically a database or external service). Spring Security uses this information to determine the user's authorization, ensuring they have the required permissions to access the requested resource. This can either be done directly in the filter chain or via the @PreAuthorize annotation at an endpoint method, which ensures that authorization checks are conducted before the execution.

By consolidating authentication and authorization into Spring Security, security rules become more transparent, reusable, and easier to manage. The application benefits from a unified security layer that is both robust and flexible, simplifying future enhancements and reducing security risks.

6.2.3 Bean Validation via Spring and Hibernate

Data beans in the QDAcity application are validated in an unstructured manner within various parts of the business logic. This approach scatters validation rules across the codebase, making it harder to manage and maintain, and increasing the likelihood of inconsistent validations.

The recommended improvement is to switch to a structured validation process of the Spring framework[6] using Jakarta Bean Validation[7] at the endpoint layer, specifically before the data enters the endpoint method. By leveraging Spring's Hibernate validator[8], the data validation can be centralized and performed as soon as the request is received by the endpoint, ensuring that only valid data proceeds further into the application.

With Spring's bean validation, annotations such as @NotNull, @Size, and @Pattern are applied directly to the fields of data beans. When a request reaches the endpoint, Spring automatically validates the incoming data against these rules before passing it into the business layer. If the data fails validation, the process is stopped, and an appropriate error response is returned to the client.

This approach ensures that data is validated consistently across all endpoints, keeping validation logic separate from business logic and making it easier to maintain. It also improves the reliability of the application by ensuring that invalid data is caught early, preventing unnecessary processing and reducing potential errors in the business layer.

[6]https://docs.spring.io/spring-framework/reference/core/validation/beanvalidation.html
[7]https://beanvalidation.org
[8]https://hibernate.org/validator/

6.2.4 CES API Calls via Spring REST Clients

At the moment, the QDAcity backend uses Google's HttpRequestFactory to send requests to the CES API. This approach is functional, but not well integrated with the Spring ecosystem, leading to potential limitations in flexibility, maintainability, and available features.

A better approach is to switch to using a Spring REST client[9], such as RestClient or WebClient, which are designed to work seamlessly within Spring applications. RestClient is a synchronous, blocking client that is simple and easy to use. It is well-suited for traditional applications where requests are handled sequentially. RestClient provides a clean API for making REST calls and integrates smoothly with Spring's dependency injection.

In contrast, WebClient is a non-blocking, reactive client that supports asynchronous operations. It is designed for modern applications that require high concurrency and performance, allowing multiple requests to be processed efficiently in parallel. WebClient leverages reactive programming and is ideal for applications that need to handle real-time data or high loads with non-blocking input and output.

Switching from HttpRequestFactory to either RestClient or WebClient improves integration with the Spring framework and makes at least the Google API Client for Java[10] dependency redundant. It provides better error handling, enhanced flexibility, and more powerful ways to handle REST API calls to the CES. The choice between RestClient and WebClient depends on whether the application benefits more from a simple, synchronous approach or a cumbersome, non-blocking solution.

[9] https://docs.spring.io/spring-framework/reference/web/webmvc-client.html
[10] https://developers.google.com/api-client-library/java

7 Conclusion

In this thesis, the migration of the backend framework for the QDAcity web application was successfully undertaken, addressing the challenges of moving from an outdated and unsupported system to a modern, long-term solution. The project involved not only a backend migration but also significant updates to the overall technology stack, including the frontend and key aspects of the system's architecture. Throughout the migration, maintaining core functionality and preserving system behavior were prioritized to ensure a seamless transition for users.

The thesis began by outlining the functional and non-functional requirements in chapter 2, focusing on key goals such as performance, security, and scalability. These requirements guided the decision-making process, ensuring that the new framework would support the evolving needs of the QDAcity platform while remaining reliable and robust.

Chapter 3 provided an overview of the existing architecture of QDAcity, identifying the legacy components in need of migration. This chapter also highlighted the challenges presented by outdated technologies and the constraints of maintaining core system functionality during the migration process. In addition, it gave a short introduction to the most important framework technologies used for QDAcity and justified the technology and product decisions.

In chapter 4, the detailed system design and implementation of several components were outlined. The engineering work included the careful migration of endpoint classes using the facade pattern (Gamma et al., 1994), allowing both old and new frameworks to coexist temporarily. The engineering approach emphasized gradual change, ensuring that the system remained operational while progressively incorporating new technologies.

The evaluation in chapter 5 critically assessed the outcomes of the migration. A performance analysis revealed a potential for improvements, while the remaining assessment demonstrated the success of the new framework in addressing the initial functional and non-functional requirements. Additionally, the updated system met the long-term goals set for QDAcity.

Finally, chapter 6 discussed the remaining tasks and future enhancements, including potential improvements in system design and further optimization efforts. These forward-looking observations set the stage for continuous development of the platform, ensuring its ability to adapt to future technical demands.

Regarding the transferability of the migration strategy, several key insights from the QDAcity migration can be applied to other software projects, although each project presents its own set of challenges based on its specific legacy architecture and technology stack. The width-first migration approach using the facade pattern, combined with the simultaneous operation of both old and new frameworks, offers a flexible and practical model for similar projects. Furthermore, the use of OpenAPI and Swagger Client to facilitate frontend integration also provides a transferable strategy for faster API migrations.

In conclusion, while the migration strategy employed for QDAcity was customized to its specific needs, many of the solutions developed are adaptable to other projects. The experience gained in this project contributes to a broader understanding of software migration processes and offers valuable lessons for the software engineering community. In particular, the insights gained from transitioning from legacy frameworks to modern solutions reflect the natural evolution of software systems, ensuring their continued relevance and adaptability in a rapidly changing technological landscape.

Appendices

A Endpoint Design and Implementation

A.1 Endpoint Processing Example

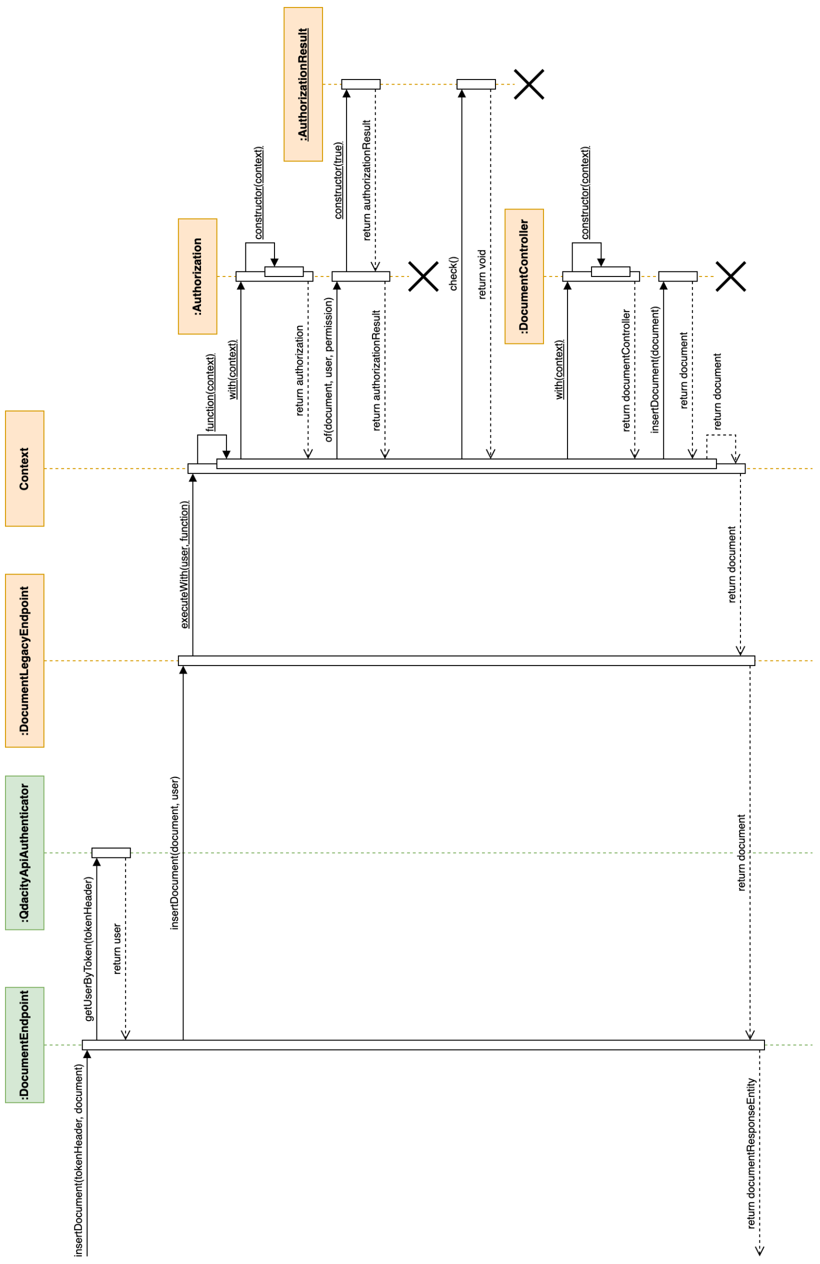


Figure 1: UML sequence diagram of the document endpoint as an example

A.2 Endpoint Class Example

Java code of the DocumentEndpoint class:

```java
@RestController
@RequestMapping("${qdacity.api.path}/documents")
@RequiredArgsConstructor
@ApiResponses(value = {
    @ApiResponse(
        responseCode = "200", description = "OK"),
    @ApiResponse(
        responseCode = "204", description = "No Content"),
    @ApiResponse(
        responseCode = "400", description = "Bad Request"),
    @ApiResponse(
        responseCode = "404", description = "Not Found"),
    @ApiResponse(
        responseCode = "500",
        description = "Internal Server Error")
})
@Tag(name = "documents")
public class DocumentEndpoint {
    private final DocumentLegacyEndpoint
        documentLegacyEndpoint;
    private final QdacityApiAuthenticator
        qdacityApiAuthenticator;

    // ...
}
```

A.3 Endpoint Method Examples

Java code of the getDocument method in the DocumentEndpoint class:

```java
@GetMapping("/{docId}")
@Operation(
    operationId = "getDocument",
    summary = "documents.getDocument",
    description = "Get document by ID",
    security = @SecurityRequirement(name = "bearerAuth"))
public ResponseEntity<BaseDocument> getDocument(
    @PathVariable(name = "docId")
        Long docId,
    @RequestHeader(name = "Authorization")
    @Parameter(hidden = true)
        String tokenHeader
) {
    final User user =
        qdacityApiAuthenticator.getUserByToken(tokenHeader);
    final BaseDocument document =
        documentLegacyEndpoint.getDocument(docId, user);
    return ResponseEntity.ok().body(document);
}
```

Java code of the insertDocument method in the DocumentEndpoint class:

```java
@PostMapping
@Operation(
    operationId = "insertDocument",
    summary = "documents.insertDocument",
    description = "Insert a new document",
    security = @SecurityRequirement(name = "bearerAuth"))
public ResponseEntity<BaseDocument> insertDocument(
    @RequestHeader(name = "Authorization")
    @Parameter(hidden = true)
        String tokenHeader,
    @RequestBody
        BaseDocument document
) {
    final User user =
        qdacityApiAuthenticator.getUserByToken(tokenHeader);
    final BaseDocument insertedDocument =
        documentLegacyEndpoint.insertDocument(document, user);
    return ResponseEntity.ok().body(insertedDocument);
}
```

B Endpoint Service Method Examples

JavaScript code of the getDocuments method in the DocumentsEndpoint class:

```
static getDocuments(projectId, projectType) {
    const apiMethodCallback = () => {
        return qdacityApiClient.then((apis) =>
            apis.documents.getDocuments);
    };
    return Promisizer.makeResponseHandlerPromise(
        apiMethodCallback,
        {
            projectId: projectId,
            projectType: projectType,
        }
    ).then((body) => body.items);
}
```

JavaScript code of the insertDocument method in the DocumentsEndpoint class:

```
static insertDocument(doc) {
    const apiMethodCallback = () => {
        return qdacityApiClient.then((apis) =>
            apis.documents.insertDocument);
    };
    return Promisizer.makeResponseHandlerPromise(
        apiMethodCallback,
        {},
        {
            requestBody: doc,
        }
    );
}
```

C Performance Analysis

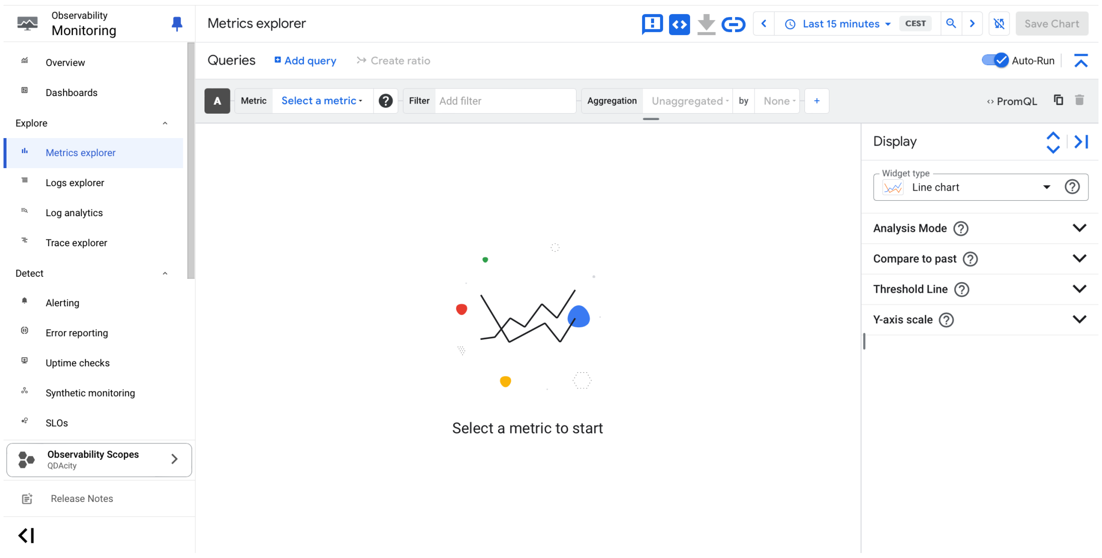


Figure 2: Metrics explorer window of the Google App Engine monitoring tool

C.1 CPU Utilization

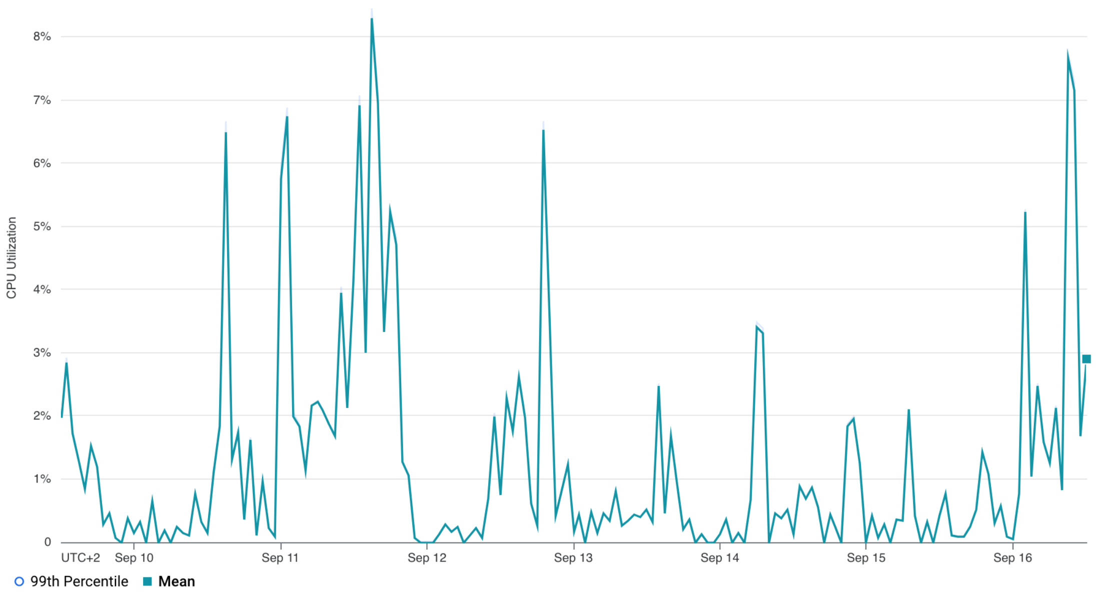


Figure 3: Line chart visualizing the CPU utilization of QDAcity version 230 based on the Google Endpoints framework

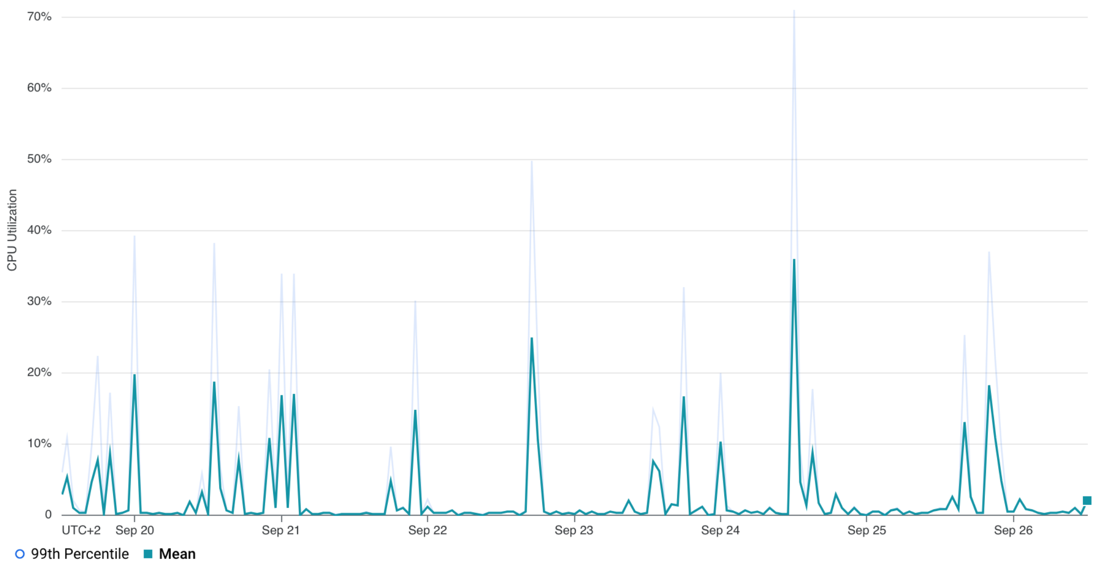


Figure 4: Line chart visualizing the CPU utilization of QDAcity versions 231 to 236 based on the Spring Boot framework

C.2 Memory Usage

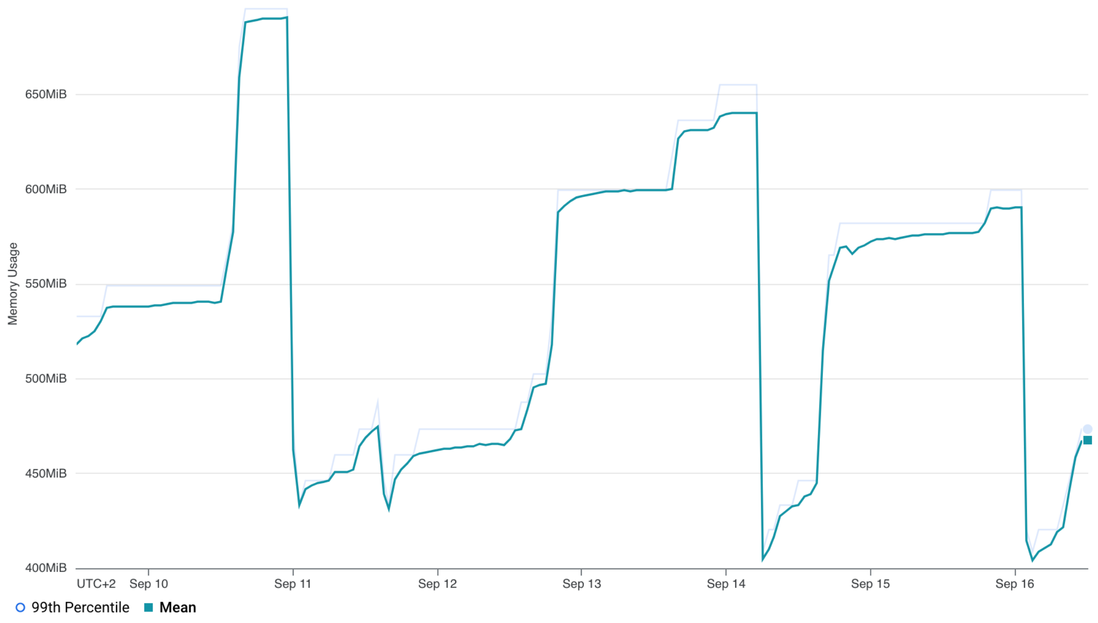


Figure 5: Line chart visualizing the memory usage of QDAcity version 230 based on the Google Endpoints framework

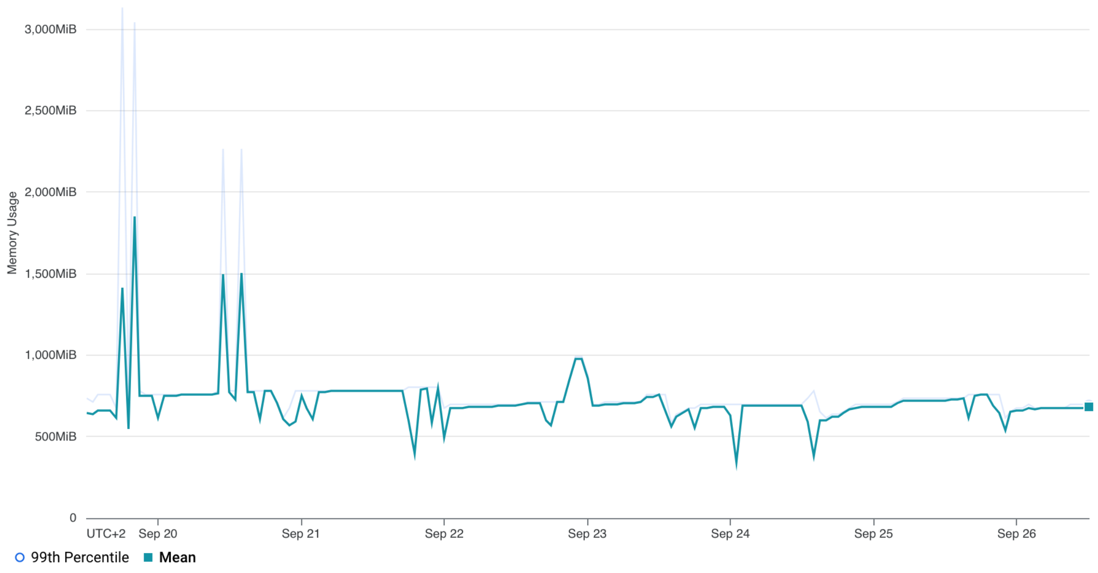


Figure 6: Line chart visualizing the memory usage of QDAcity versions 231 to 236 based on the Spring Boot framework

C.3 Response Count

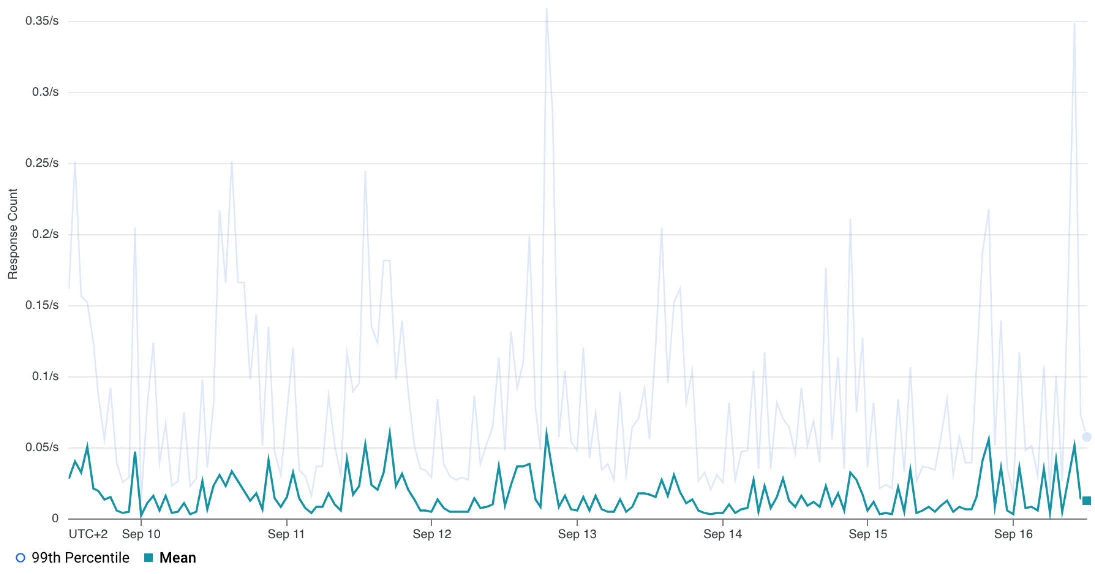


Figure 7: Line chart visualizing the response count of QDAcity version 230 based on the Google Endpoints framework

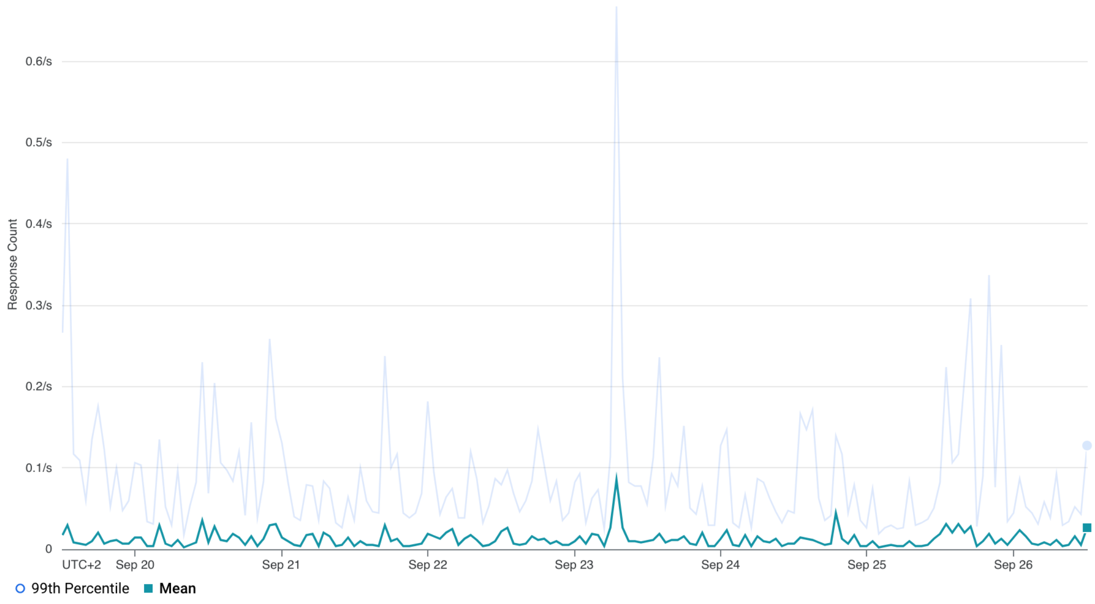


Figure 8: Line chart visualizing the response count of QDAcity versions 231 to 236 based on the Spring Boot framework

C.4 Response Latency

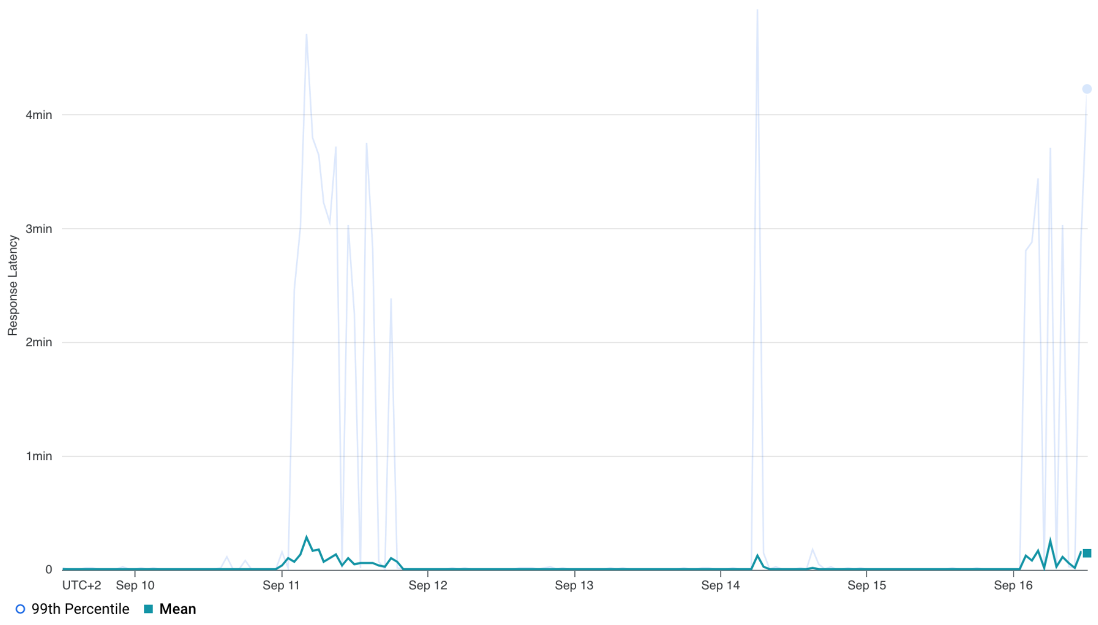


Figure 9: Line chart visualizing the response latency of QDAcity version 230 based on the Google Endpoints framework

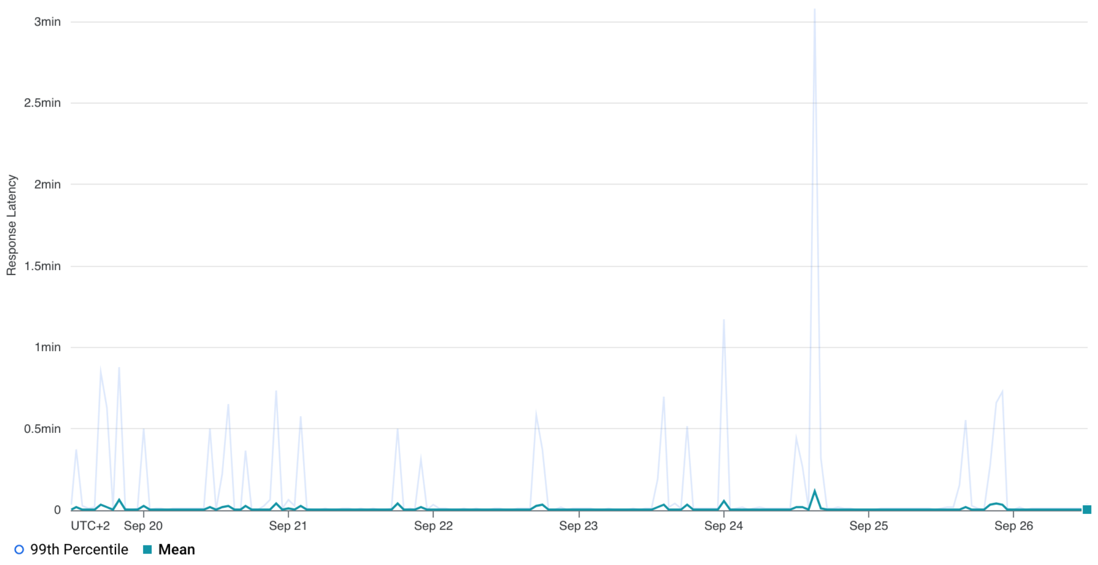


Figure 10: Line chart visualizing the response latency of QDAcity versions 231 to 236 based on the Spring Boot framework

References

Alexenko, T., Jenne, M., Roy, S. D., & Zeng, W. (2010). Cross-site request forgery: Attack and defense. *2010 7th IEEE Consumer Communications and Networking Conference*, 1–2. https://doi.org/10.1109/CCNC.2010.5421782

Fowler, M. (2003). *Patterns of enterprise application architecture* (1st ed.). Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software* (1st ed.). Addison-Wesley Professional.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 1060–1076. https://doi.org/10.1109/PROC.1980.11805

Richards, M. (2015). *Software architecture patterns: Understanding common architecture patterns and when to use them* (1st ed.). O'Reilly Media.

SOPHISTen. (2024). Master: Schablonen für alle fälle. https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/Wissen_for_free/MASTeR-Broschuere_Int/MASTeR_Broschuere_6-Auflage_31-07-2024_AvP_V4.pdf