Bachelorarbeit im Studiengang Medieninformatik

How does an Event Sourcing architecture compare to CRUD systems with an independent audit log, when it comes to scalability, performance and traceability?

Vorgelegt von Lukas Karsch

an der Hochschule der Medien Stuttgart am 02.03.2026

zur Erlangung des akademischen Grades eines Bachelor of Science

Erstprüfer: Prof. Dr. Tobias Jordine

Zweitprüfer: Felix Messner

Ehrenwörtliche Erklärung

Hiermit versichere ich, Lukas Karsch, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: "How does an Event Sourcing architecture compare to CRUD systems with an independent audit log, when it comes to scalability, performance and traceability?" selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Ebenso sind alle Stellen, die mit Hilfe eines KI-basierten Schreibwerkzeugs erstellt oder überarbeitet wurden, kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.“ Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Lukas Karsch

Contents

List of Figures

List of Tables

Listings

Glossary

ACID Atomicity, Consistency, Isolation, Durability. 5, 7

Anemic Domain Model The objects describing the domain only hold data, no logic. 3–5

API API stands for *Application Programming Interface*. It describes the public interface of a module or service, often exposed over a network. 12, 14

Atomicity Atomicity means that an action is either fully executed or not at all. Atomic operations make sure the application is not left in an invalid state (Bernstein & Newcomer, 2009, p. 10). 6

BASE Basically Available, Soft State, Eventual Consistency. 7

Contract Test A contract test verifies that services implement a shared interface by testing their interactions against an explicitly defined contract. 11

CQRS Command Query Responsibility Segregation. 6, 8, 11, 12, 15, 32

CQS Command And Query Separation. 6

CRUD Create Read Update Delete. 5, 11, 12, 14, 27

DAO Data Access Object. 3

DDD Domain Driven Design. 4, 15, 16

Docker Open platform for developing, shipping and running containerized applications. viii, 20

Dockerfile A text document containing a series of instructions used to assemble a Docker image. 20

DSL Data Specific Language. 19

DTO Data Transfer Object. 6

ES Event Sourcing. 11, 12

GPath Expression A Groovy-based path language for navigating and querying nested object graphs (such as JSON or XML) using concise, expressive selectors and closures (Apache Groovy project, n.d.-a). 19

Groovy A dynamic JVM language that extends Java with concise syntax and powerful features such as closures, making it well suited for scripting, DSLs, and test code (Apache Groovy project, n.d.-b). 19

HATEOAS Hypermedia as the engine of application state. 2

HTML HyperText Markup Language. 3

HTTP HTTP stands for *Hypertext Transfer Protocol*. It is a protocol used in internet communication and was defined in RFC 2616 (The Internet Society, 1999). viii, 2, 12, 19, 22, 23

JMX Java Management Extensions. 19

JPA Jakarta Persistence API. 14, 24–26, 30

JPQL Java Persistence Query Language. 14

JSON JavaScript Object Notation. 3, 14, 28, 29, 32

ORM Object-relational Mapper. 14

POJO Plain Old Java Object. 15

REST REST stands for *Representational State Transfer*. It is an architectural style for distributed hypermedia systems. 2

REST Assured A library for testing servers. 18, 23

Rich Domain Model Objects incorporate both data and the behavior or rules that govern that data. 4, 5

Testcontainer Testcontainers are a way to declare infrastructure dependencies as code using Docker (Testcontainers, n.d.). 23

URI Uniform Resource Identifier. 2

VU Virtual User. 21

WWW World Wide Web. 2

XML Extensible Markup Language. 3, 8, 14

Chapter 1

Introduction

1.1 Motivation

1.2 Research question(s)

1.3 Goals and non goals

1.4 Structure of the paper

Chapter 2

Basics

2.1 WWW, Web APIs, REST

The World Wide Web (WWW) is a connected information network used to exchange data. Resources are can be accessed via Uniform Resource Identifiers (URIs) which are transferred using formats like JSON or HTML via protocols like HTTP. HTTP is a stateless protocol based on a request-response structure. It supports standardized request types, such as `GET` and `POST`, which convey a semantic meaning (Jacobs & Walsh, 2004).

Web APIs are interfaces that enable applications to communicate. They use HTTP as a network-based API (Fielding, 2000, p. 138). Modern APIs typically follow REST principles. REST stands for "Representational State Transfer" and describes an architectural style for distributed hypermedia systems (Fielding, 2000, p. 76).

REST APIs adhere to principles derived from a set of constraints imposed by the HTTP protocol, for example. One such constraint is "stateless communication": Communication between clients and the server must be *stateless*, meaning the client must provide all the necessary information for the server to fully understand the request.

Furthermore, every resource in REST applications must be addressable via a unique ID, which can then be used to derive a URI to access the resource. Below are some examples for resources and URIs which could be derived from them:

- Book; ID=1; URI=`http://example.com/books/1`
- Book; ID=2; URI=`http://example.com/books/2`
- Author; ID=100; URI=`http://example.com/authors/100`

The "Hypermedia as the engine of application state (HATEOAS)" principle states that resources should be linked to each other. Clients should be able to

control the application by following a series of links provided by the server (Tilkov, 2007).

Every resource must support the same interface, usually HTTP methods (GET, POST, PUT, etc.) where operations on the resource correspond to one method of the interface. For example, a POST operation on a customer might map to the `createCustomer()` operation on a service.

Resources are decoupled from their representations. Clients can request different representations of a resource, depending on their needs (Tilkov, 2007): a web browser might request HTML, while another server or application might request XML or JSON.

2.2 Layered Architecture Foundations

Layered Architecture is the most common architecture pattern in enterprise applications. Applications following a layered architecture are divided into *horizontal layers*, with each layer performing a specific role. A standard implementation consists of the following layers:

- Presentation: Handles requests and displays data in a user interface or by turning it into representations (e.g. JSON)
- Business: Encapsulates business logic
- Persistence: Persists data by interacting with the underlying persistence technologies (e.g. SQL databases)
- Database

A key concept in this design is layers of isolation, where layers are "closed", meaning a request must pass through the layer directly below it to reach the next, ensuring that changes in one layer do not affect others.

In a layered application, data flows downwards during request handling and upwards during the response: a request arrives in the presentation layer, which delegates to the business layer. The business layer fetches data from the persistence layer which holds logic to retrieve data, e.g. by encapsulating SQL statements.

The database responds with raw data, which is turned into a Data Access Object (DAO) by the persistence layer. The business layer uses this data to execute rules and make decisions. The result will be returned to the presentation layer which can then wrap the response and return it to the caller. (Richards, 2015)

The data in layered applications is often times modeled in an *anemic* way. In an Anemic Domain Model, business entities are treated as only data. They are objects which contain no business logic, only getters and setters. Business logic is entirely contained in the business (or "service") layer. Fowler (2003) describes this as an object-oriented *antipattern*.

2.3 Domain Driven Design

Domain Driven Design (DDD) is a different architectural approach for applications. It differs from layered architecture primarily in the way the domain is modelled and the responsibilities of application services.

The core idea of DDD is that the primary focus of a software project should not be the underlying technologies, but the domain. The domain is the topic with which a software concerns itself. The software design should be based on a model that closely matches the domain and reflects a deep understanding of business requirements. (Evans, 2004, pp. 8, 12)

This domain model is built from a *ubiquitous language* which is a language shared between domain experts and software experts. This ubiquitous language is built directly from the real domain and must be used in all communications regarding the software. (Evans, 2004, pp. 24–26)

The software must always reflect the way that the domain is talked about. Changes to the domain and the ubiquitous language must result in an immediate change to the domain model.

When modeling the domain model, the aim should not be to create a perfect replica of the real world. While it should carefully be chosen, the domain model is artificial and forms a selective abstraction which should be chosen for its utility. (Evans, 2004, pp. 12, 13)

While Layered Architecture organizes code into technical tiers and is typically built on Anemic Domain Models, often resulting in the *big ball of mud* antipattern (Richards, 2015, p. V), DDD demands a Rich Domain Model where objects incorporate both data and the behavior or rules that govern that data. The code is structured semantically into bounded context and modules which are chosen to tell the "story" of a system rather than its technicalities. (Evans, 2004, p. 80)

Entities (also known as reference objects) are domain elements fundamentally defined by a thread of continuity and identity rather than their specific attributes. Entities must be distinguishable from other entities, even if they share the same characteristics. To ensure consistency and identity, a unique identifier is assigned to entities. This identifier is immutable throughout the object's life. (Evans, 2004, pp. 65–69)

Value Objects are elements that describe the nature or state of something and have no conceptual identity of their own. They are interesting only for their characteristics. While two entities with the same characteristics are considered as different from each other, the system does not care about "identity" of a value object, since only its characteristics are relevant. Value objects should be used to encapsulate concepts, such as using an "Address" object instead of distinct "Street" and "City" attributes. Value objects should be immutable. They are never modified, instead they are replaced entirely when a new value is required. (Evans, 2004, pp. 70–72)

Using a Rich Domain Model does not mean that there should be no layers, the opposite is the case. Evans (2004) advocates for using layers in domain driven designs. He proposes the following layers: (Evans, 2004, p. 53)

- Presentation: Presents information and handles commands
- Application Layer: Coordinates app activity. Does not hold business logic, but delegate tasks and hold information about their progress
- Domain Layer: Holds information about the domain. Stateful objects (rich domain model) that hold business logic and rules
- Infrastructure layer: Supports other layers. Handles concerns like communication and persistence

Evans (2004, p. 75) points out that in some cases, operations in the domain can not be mapped to one object. For example, transferring money does conceptually not belong to one bank account. In those cases, where operations are important domain concepts, domain services can be introduced as part of model-driven design. To keep the domain model rich and not fall back into procedural style programming like with an Anemic Domain Model, it is important to use services only when necessary. Services are not allowed to strip the entities and value objects in the domain of behavior. According to Evans, a good domain service has the following characteristics:

- The operation relates to a domain concept which would be misplaced on an entity or a value object
- The operation performed refers to other objects in the domain
- The operation is stateless

2.4 CRUD architecture

Layered architectures are the standard for data-oriented enterprise applications. These applications mostly follow a CRUD architecture. CRUD is an acronym coined by Martin (1983) that stands for "Create, Read, Update, Delete". These four actions can be applied to any record of data.

The state of domain objects in a CRUD architecture is often mapped to normalized tables on a relational database, though other storage mechanisms maybe used. The application acts on the current state of the data, with all actions (reads and writes) acting on the same data.

ACID (Atomicity, Consistency, Isolation, Durability) are an important feature of CRUD applications. They can be guaranteed using transactions, ensuring that

data stays consistent and operations are atomic. (Bernstein & Newcomer, 2009, pp. 10, 11)

Databases in CRUD systems are typically normalized. Normalization is a process of organizing data into separate tables, removing redundancies and creating relationships through "foreign keys". It is the best practice for relational databases. There are several normal forms that can be achieved, each form building on the previous one: to achieve the second normal form, the first normal form has to be achieved first. (Martin, 1983, p. 203)

- 1NF (First Normal Form): Each table cell contains a single (atomic) value, every record is unique
- 2NF (Second Normal Form): Remove partial dependencies by requiring that all *non-key* columns are fully dependent on the primary key
- 3NF (Third Normal Form): Removes transitive dependencies by requiring that non-key columns depend *only* on the primary key
- Further Normal Forms (4NF, 5NF): Require a table can not be broken down into smaller tables without losing data

2.5 CQRS Architecture

Command Query Responsibility Segregation (CQRS) is an architectural pattern based on the fundamental idea that the models used to update information should be separate from the models used to read information. This approach originated as an extension of Bertrand Meyer's Command And Query Separation (CQS) principle, which states that a method should either perform an action (a command) or return data (a query), but never both. (Meyer, 2006, p. 148)

CQRS is different from CQS in the fact that in CQRS, objects are split into two objects, one containing commands, one containing queries. (Young, 2010, p. 17)

CQRS applications are typically structured by splitting the application into two paths:

- Command Side: Deals with data changes and captures user intent. Commands tell the system what needs to be done rather than overwriting previous state. Commands are validated by the system before execution and can be rejected. (Young, 2010, pp. 11, 12)
- Read Side: Strictly for reading data. The read side is not allowed to modify anything in the primary data store. The read side typically stores Data Transfer Objects (DTOs) in its own data store that can directly be returned to the presentation layer. (Young, 2010, p. 20)

In a CQRS architecture, the read side typically updates its data asynchronously by consuming notifications or events generated by the write side. Because the models for updating and reading information are strictly separated, a synchronization mechanism is required to ensure the read store eventually reflects the changes made by commands. This usually leads to stale data on the read side.

Each read service independently updates its model by consuming notifications or events published by the write side, allowing the read model to store optimized, denormalized views on the data. (Young, 2010, p. 23)

2.6 (Eventual) Consistency

Gray et al. (1996) explain that large-scale systems become unstable if they are held consistent at all times according to ACID principles. This is mostly due to the large amount of communication necessary to handle atomic transactions in distributed systems. To address these issues, modern distributed systems often adopt the BASE (Basically Available, Soft State, Eventual Consistency) model which explicitly trades off isolation and strong consistency for availability. Eventually consistent systems are allowed to exist in a so-called "soft state" which eventually converges through the use of synchronization mechanisms over time rather than being strongly consistent at all times. (Braun et al., 2021; Vogels, 2009) This creates an inconsistency window in which data is not consistent across the system. During this window, stale data may be read. (Vogels, 2009)

2.7 Event Sourcing and event-driven architectures

Event driven architecture is a design paradigm where systems communicate via the production and consumption of events. Events are records of changes in the system's domain. (Michelson, 2006) This approach allows for a high degree of loose coupling, as the system publishing an event does not need to know about the recipient(s) or how they will react. These architectures offer excellent horizontal scalability and resilience, as individual system components can fail or be updated without bringing down the entire network. (Fowler, 2005)

Event Sourcing is an architectural pattern within the landscape of event driven architectures. Event-sourced systems ensure that all changes to a system's state are captured and stored as an ordered sequence of domain events. Unlike traditional persistence models that overwrite data and store only the most recent state, event sourcing maintains an immutable record of every action taken over time. These events are persisted in an append-only event store, which serves as the principal source of truth from which the current system state can be derived. (Kleppmann, 2017, pp. 457, 458)

The current state of any entity in such a system can be rebuilt by replaying the history of events from the log, starting from an initial blank state. (Fowler, 2005) To address the performance costs of replaying thousands of events for every request, developers implement projections or materialized views, which are read-only, often denormalized versions of the data optimized for specific queries. (Malyi & Serdyuk, 2024; Kleppmann, 2017, pp. 461, 462) This separation of concerns is frequently managed by pairing event sourcing with the Command Query Responsibility Segregation (CQRS) pattern, which physically divides the data structures used for reading from those used for writing state changes. (Young, 2010, p. 50)

2.8 Traceability and auditing in IT systems

Traceability and auditing are legal requirements across various sectors, as they are derived from federal laws and regulations intended to protect the integrity and confidentiality of sensitive data. Organizations implement these mechanisms to stay compliant with mandates that require a verifiable, time-sequenced history of system activities to support oversight and forensic reviews. In the U.S. financial sector, for example, 17 CFR § 242.613 requires the establishment of a consolidated audit trail to track the complete lifecycle of securities orders, documenting every stage from origination and routing to final execution. (U.S. Securities and Exchange Commission, 2012)

2.8.1 Audit Logs

An audit log (often called audit trail) is a chronological record which provides evidence of a sequence of activities on an entity. (Committee on National Security Systems, 2010) In information security, the audit log stores a record of system activities, enabling the reconstruction of events. (ATIS Committee, 2013) A trustworthy audit log in a system can guarantee the principle of traceability which states that actions can be tracked and traced back to the entity who is responsible for them. (Joint Task Force Interagency Working Group, 2020, p. 266)

Fowler (2004) describes an audit log as simple and effective way of storing temporal information. Changes are tracked by writing a record indicating *what* changed *when*. A basic implementation of an audit log can have many forms, for example a text file, database tables or XML documents. Fowler also mentions that while the audit log is easy to write, it is harder to read and process. While occasional reads can be done by eye, complex processing and reconstruction of historical state can be resource-intensive.

2.8.2 Event Streams as a Basis for Traceability

While traditional audit logs are often implemented as secondary systems that capture state changes, event-driven architectures, such as those utilizing Event Sourcing, turn an event stream into the primary source of truth. In this context, an event stream is not just a diagnostic tool but an exact, chronological sequence of intent-driven records.

As established in section 2.7, every state change is captured as a discrete event. Because these events are immutable and append-only, they provide a natural foundation for the principle of traceability. Unlike traditional "state-based" auditing, where the system might only record that a value changed from A to B, an event stream captures the specific domain context. Tthe *intent* behind a change is semantically conveyed through the event type. For example, while a traditional audit log might simply record a status update to `CLOSED`, an event-sourced system distinguishes between an `AccountDeletedByUser` event and an `AccountTerminatedForInactivity` event. This inherent metadata provides an exhaustive audit trail without the need for additional logging logic.

Fowler (2005) notes that because the event log is complete, the system can perform *Temporal Queries*, effectively "time-traveling" to reconstruct the exact state of the system at any historical checkpoint. This makes event streams particularly robust for forensic reviews and regulatory compliance, as they eliminate the "information loss" associated with traditional database overwrites. In the context of the legal requirements discussed in the previous section, the event stream serves as a sequence of actions that satisfies the need for a verifiable, time-sequenced history.

2.8.3 Rebuilding state from an audit log and an event stream

2.9 Scalability of systems

Chapter 3

Related Work

Chapter 4

Proposed Method

This thesis aims to provide a fair, quantitative comparison of CRUD and CQRS / ES architectures. To achieve this, the architectures should be applied not only to the same domain, but to the exact same requirements. The implementations can then be tested against the same contract tests.

This chapter will first present the requirements for the actual application, then outline metrics and comparison methods.

4.1 Project requirements

The applications will implement a course enrollment and grading system which might for example be used in universities. Core features include:

- Professors can create courses and lectures
- Students can enroll and disenroll from lectures
- Professors can enter grades
- Students can view their current and past lectures
- Students can view their credits

4.1.1 Entities

Two types of users exist in the domain: professors and students. Their personal information is not relevant for this thesis, which is why only their first and last name are stored for presentation reasons. The student additionally has a semester.

Professors can create courses. Courses have a name, a description, an amount of credits they yield, a minimum amount of credits required to enroll and can have a set of courses as prerequisites.

Courses are the "blueprints" for lectures. Lectures are the "implementation" of a course for a semester. Each lecture created from a course yields the course's amount of credits and has the requirements specified by the course. Lectures have a lifecycle: they can be in draft state, open for enrollment, in progress, finished or archived. A lecture has a list of time slots and a maximum amount of students that can enroll.

A lecture can have several assessments. Each assessment has a type. The professor can enter grades for a student and an assessment. Grades are integers in the range of 0 to 100. Credits are awarded to a student as soon as they completed all assessments for a lecture with a passing grade (grade higher than 50).

4.1.2 Business rules

Relationships and business rules in this system are deliberately chosen complex, involving many relationships between entities and intricate validation rules. This approach was adopted in order to be able to make realistic assumptions about the research question by evaluating a project that closely resembles complex, real-world scenarios.

- Existence checks: any requests including references to entities will fail if the references entities do not exist.
- Requests leading to conflicts, for example creating a lecture with overlapping time slots, will fail.
- When a student tries enrolling to a lecture which is already full, they will be put on a waitlist.
- When a student disenrolls from a lecture, the next eligible student (higher semesters are preferred) will be enrolled.
- Actions on a lecture can only be done during the appropriate lifecycle state (enrolling only when the lifecycle is "open for enrollment", grades can only be assigned when the lecture is "finished").

4.1.3 Contract Tests

To ensure both implementations adhere to the business rules, an extensive test suite was set up. While the internals of the implementations are vastly different architecturally and conceptually, they both have the same public API. This makes it possible to run the same test suite on both apps by sending HTTP requests and verifying their responses. The test suite includes integration tests for all API endpoint covering both regular and edge-case (error) scenarios to ensure that the CRUD and ES-CQRS application exhibit identical state transitions and error behaviors. section 5.1 outlines the implementation of those tests in detail.

4.2 Performance

4.3 Scalability or flexibility (TODO)

4.4 Traceability

4.5 Tech Stack

4.5.1 SpringBoot

SpringBoot [1] is an open-source, opinionated framework for developing enterprise Java applications. It is based on Spring Framework [2], which is a platform aiming to make Java development "quicker, easier, and safer for everybody" (Broadcom, Inc., 2026c). At Spring Framework's core is the Inversion of Control (IoC) container. The objects managed by this container are referred to as *Beans*. While the term originates from the JavaBeans specification, a standard for creating reusable software components, Spring extends this concept by taking full responsibility for the lifecycle and configuration of these objects (Walls, 2016, Chapter 1.1). Instead of a developer manually instantiating classes using the `new` operator, the container "injects" required dependencies at runtime. This process is known as Dependency Injection. (Deinum et al., 2023, Chapter 1). Spring offers support for several programming paradigms: reactive, event-driven, microservices and serverless. (Broadcom, Inc., 2026c)

SpringBoot builds on top of the Spring platform by applying a "convention-over-configuration" approach, intended to minimize the need for configuration. In a 2023 survey by JetBrains, SpringBoot was the most popular choice of web framework. (JetBrains, 2023)

Spring Boot starters are specialized dependency descriptors designed to simplify dependency management by aggregating commonly used libraries into feature-defined packages. Rather than requiring developers to manually identify and maintain a list of individual group IDs, artifact IDs, and compatible version numbers for every necessary library, starters use transitive dependency resolution to pull in all required components under a single entry. To quickly bootstrap a web application, a developer can simply add the `spring-boot-starter-web` dependency to their Maven or Gradle build file. By requesting this specific functionality, Spring Boot automatically includes essential dependencies such as Spring MVC, Jackson for JSON processing, and an embedded Tomcat server, ensuring that all included libraries have been tested together for compatibility. This approach shifts the developer's focus from managing individual JAR files to simply defining the high-level

[1] SpringBoot
[2] Spring Framework

capabilities the application requires, minimizing configuration overhead and reducing risk of version mismatches. (Walls, 2016, Chapter 1.1.2)

4.5.2 JPA

Jakarta Persistence API (JPA) [3], formerly Java Persistence API is a Java specification which provides a mechanism for managing persistence and object-relational mapping (ORM). Object-relational Mappers (ORMs) act as a brige between the relational world of SQL databases and the object-oriented world of Java.

Instead of writing SQL to create the database schema, entities can be described using special Java classes (defined by annotations or XML configurations) which can be mapped to an SQL schema. JPA allows querying the database for these entities in a type-safe way by providing a range of helpful query methods on JPA repositories, for example `findAll()` or `findById(UUID id)`. This removes the need to write "low-level", database-specific SQL for basic CRUD operations. Complex data retrieval is also possible with JPA using the Java Persistence Query Language (JPQL), which is an object-oriented, database-agnostic query language.

When using JPA with SpringBoot by including the `spring-boot-starter-data-jpa` dependency, *Hibernate* [4] is used as implementation of the JPA standard. (Bauer, 2016, Chapter 1)

4.5.3 PostgreSQL

PostgreSQL [5] is an open-source relational database which has been in active development for over 35 years. It is designed for a wide range of workloads and can handle many tasks thanks to its extensibility and large suite of extensions, such as the popular PostGIS extension for storing and querying geospatial data. (PostGIS PSC, 2023) As of January 2026, PostgreSQL 18 is the latest version. (PostgreSQL Global Development Group, 2026)

4.5.4 Jackson

Jackson [6] is a high-performance, feature-rich JSON processing library for Java. It is the default JSON library used within the Spring Boot ecosystem. Its primary purpose is to provide a seamless bridge between Java objects and JSON data through three main processing models: the Streaming API for incremental parsing, the Tree Model for a flexible node-based representation, and the most commonly used Data Binding module. This data binding capability allows developers to automatically

[3] JPA
[4] Hibernate
[5] PostgreSQL
[6] Jackson

convert (*marshal*) Java Plain Old Java Objects (POJOs) into JSON and vice versa (*unmarshal*) with minimal configuration. Beyond its speed and efficiency, Jackson is highly extensible, offering modules to handle complex Java types like Java 8 Date/Time and Optional classes. Jackson also supports various other data formats such as XML, YAML and CSV. (FasterXML, 2025; Oracle, n.d.)

4.5.5 Axon

Axon Framework [7] is an open-source Java framework for building event-driven applications. Following the CQRS and event-sourcing pattern, Commands, Events and Queries are the three core message types any Axon application is centered around. Commands are used to describe an intent to change the application's state. Events communicate a change that happened in the application. Queries are used to request information from the application.

Axon also supports Domain Driven Design by providing tools to manage entities and domain logic. (Axoniq, 2025g, 2025i)

Axon Server [8] is a platform designed specifically for event-driven systems. It functions as both a high-performance Event Store and a dedicated Message Router for commands, queries, and events. By bundling these responsibilities into a single service, Axon Server replaces the need for separate infrastructures such as a relational database for events and a message broker like Kafka or RabbitMQ for communication. Axon Server is designed to seamlessly integrate with Axon Framework. When using the Axon Server Connector, the application automatically finds and connects to the Axon Server. It is then possible to use the Axon server without further configuration. (Axoniq, 2025a, 2025h)

Command dispatching

Command dispatching is the starting point for handling a command message in Axon. Axon handles commands by routing them to the appropriate command handler. The command dispatching infrastructure can be interacted with using the low-level `CommandBus` and a more convenient `CommandGateway` which is a wrapper around the `CommandBus`.

`CommandBus` is the infrastructure mechanism responsible for finding and invoking the correct command handler. At most one handler is invoked for each command; if no handler is found, an exception is thrown.

Using `CommandGateway` simplifies command dispatching by hiding the manual creation of `CommandMessages`. The gateway offers two main methods for synchronous and asynchronous patterns. The `send` method returns a `Completable-`

[7] Axon Framework
[8] Axon Server

`Future`, which is an asynchronous mechanism in Java. If the thread needs to wait for the command result, the `sendAndWait` method can be used.

In general, a handled command returns `null`, if handling was successful. Otherwise, a `CommandExecutionException` is propagated to the caller. While returning values from a command handler is not forbidden, it is used sparsely as it contradicts with CQRS semantics. One exception: command handlers which *create* an aggregate typically return the aggregate identifier. (Axoniq, 2025b, 2025f)

Query Handling

Before a query is handled, Axon dispatches it through its messaging infrastructure. Just like the command infrastructure, Axon offers a low-level `QueryBus` which requires manual query message creation and a more high-level `QueryGateway`.

In contrast to command handling, multiple query handlers can be invoked for a given query. When dispatching a query, callers can decide whether they want a single result or results from all handlers. When no query handler is found, an exception is thrown.

The `QueryGateway` includes different dispatching methods. For regular "point-to-point" queries, the `query` method can be used. Subscription queries are queries where callers expect an initial result and continuous updates as data changes. These queries work well with reactive programming. For large result sets, streaming queries should be used. The response returned by the query handler is split into chunks and streamed back to the caller. All query methods are asynchronous by nature and return Java's `CompletableFuture`. (Axoniq, 2025k)

Aggregates

An aggregate is a core concept of Domain Driven Design (DDD). In Axon, an aggregate defines a consistency boundary around domain state and encapsulates business logic. Aggregates are the primary place where domain invariants are enforced and where commands that intend to change domain state are handled.

Aggregates define command handlers using methods or constructors annotated with `@CommandHandler`. These handlers receive commands and decide whether they are valid according to domain rules. If a command is accepted, the aggregate emits one or more domain events describing *what* happened. Command handlers are responsible only for decision-making; they must not directly mutate the aggregate's state. Instead, all state changes must occur as a result of applying events.

Every aggregate is typically annotated with `@Aggregate` and must declare exactly one field annotated with `@AggregateIdentifier`. This identifier uniquely identifies the aggregate instance. Axon uses it to route incoming commands to the correct aggregate and to load the corresponding event stream when rebuilding aggregate state.

By default, Axon uses event-sourced aggregates. This means that aggregates are not persisted as a snapshot of their fields. Instead, their current state is reconstructed by replaying all previously stored events. Methods annotated with `@EventSourcingHandler` are called by Axon during this replay process to update the aggregate's internal state based on event data. Since events represent facts that already occurred, event sourcing handlers must not contain business logic or make decisions.

Axon also supports multi-entity aggregates. In this model, an aggregate may contain child entities that participate in command handling. Such entities are registered using `@AggregateMember`, and each entity must define a unique identifier annotated with `@EntityId`. Based on this identifier, Axon is able to route commands to the correct entity instance within the aggregate. (Axoniq, 2025j)

External Command Handlers

Often, command handling functions are placed directly inside the aggregate. However, this is not required and in some cases it may not be desirable or possible to directly route a command to an aggregate. Thus, any object can be used as a command handler by including methods annotated with `@CommandHandler`. One instance of this command handling object will be responsible for handling *all* commands of the command types it declares in its methods.

In these external command handlers, aggregates can be loaded manually from Axon's repositories using the aggregate's ID. Afterward, the `execute` function can be used to execute commands on the loaded aggregate. (Axoniq, 2025c)

Events

Event handlers are methods annotated with `@EventHandler` which react to occurrences within the app by handling Axon's event messages. Each event handler specifies the types of events it is interested in. When no handler for a given event type exists in the application, the event is ignored. (Axoniq, 2025e)

Axon's `@EventBus` is the infrastructure mechanism dispatching events to the subscribed event handlers. Event stores offer these functionalities and additionally persist and retrieve published events. (Axoniq, 2025d)

Event processors take care of the technical part aspects of event processing. Axon's `EventBus` implementations support both subscribing and tracking event processors. (Axoniq, 2025d) Subscribing event processors subscribe to a message source, which delivers (pushes) events to the processor. The event is then processed in the same thread that published the event. This makes subscribing event processors suitable for real-time updates of models. However, they can only be used to receive current events and do not support event replay. Additionally, as they run on the same thread, they can not be parallelized. (Axoniq, 2025o)

Tracking event processors, which a type of streaming event processors, read (pull) events to be processed from an event source. They run decoupled from the publishing thread, making them parallelizable. These event processors use tracking tokens track their position in the event stream. Tracking tokens can be reset and events can be replayed and reprocessed. Tracking event processors are the default in Axon and recommended for most ES-CQRS use cases. (Axoniq, 2025n)

Subscribing event processors can be configured using SpringBoot's `application.properties` file or through Java configuration classes.

Sagas

In Axon, Sagas are long-running, stateful event handlers which not just react to events, but instead manage and coordinate business transactions. For each transaction being managed, one instance of a Saga exists. A Saga, which is a class annotated with `@Saga` has a lifecycle that is started by a specific event when a method annotated with `@StartSaga` is executed. The lifecycle may be ended when a method annotated with `@EndSaga` is executed; or conditionally using 'SagaLifecycle.end()'. A Saga usually has a clear starting point, but may have many different ways for it to end. Each event handling method in a Saga must additionally have the `@SagaEventHandler` annotation. (Axoniq, 2025m)

The way Sagas manage business transactions is by sending commands upon receiving events. They can be used when workflows across several aggregates should be implemented; or to handle long-running processes that may span over any amount of time. (Axoniq, 2025m) For example, the lifecycle of an order, from being processed, to being shipped and paid, is a process that usually takes multiple days. A use case like this is typically implemented using Sagas.

A Saga is associated with one or more association values, which are key-value pairs used to route events to the correct Saga instance. A `@StartSaga` method together with the `@SagaEventHandler(associationProperty="aggregateId")` automatically associates the Saga with that identifier. Additional associations can be made programmatically, by calling 'SagaLifecycle.associateWith()'. Any matching events are then routed to the Saga. (Axoniq, 2025l)

For example, a Saga managing an order's lifecycle may be started by an `@OrderPlaced` event and associated with the `orderId`. It can then issue a `CreateInvoiceCommand` using an `invoiceId` generated inside of the event handler. It then associates itself with this ID to be notified of further events regarding this invoice, such as an `InvoicePaidEvent`.

4.5.6 Testing

To ensure functionality of the applications, unit and integration tests were implemented using various testing libraries like JUnit as the testing platform, REST As-

sured for making and asserting HTTP calls, Mockito for unit testing and ArchUnit for architecture tests. This section describes all mentioned technologies.

JUnit [9] is an open-source testing framework for Java. It offers a structured way of writing tests, driven by lifecycle methods like `beforeEach` or `afterAll`. Tests are annotated with `@Test`. They can also be parametrized and run repeatedly. Results can be asserted using assertion methods like `assertTrue()`. ("JUnit User Guide", n.d.)

REST Assured [10] is a Java library that provides a highly fluent DSL for testing and validating REST APIs in a readable, chainable style. It allows complex assertions to be written inline using Groovy expressions, making it easy to deeply verify JSON responses beyond simple field checks. (Johan Haleby, n.d.)

The below code example shows how one might use a Groovy expression to find and validate a path in the returned JSON object:

```
RestAssured.when()
    // omitted request
    .then()
    .body(
        "data.grades.find { it.combinedGrade == 0 }.credits",
        equalTo(0)
    );
```

Listing 4.1: Validating JSON path using Rest Assured

Here, the path `data.grades` of the returned JSON object is expected to be an array. The array is filtered using a GPath expression with a closure to find the first entry where `combinedGrade` equals 0. Then, this entry's `credits` field is extracted and validated using the `equalTo(0)` matcher.

4.5.7 SpringBoot Actuator

Spring Boot Actuator [11] is a tool designed to help monitor and manage Spring Boot applications running in a production environment. It provides several built-in features that allow developers to check the status of the application, gather performance data, and track HTTP requests. These features can be accessed using either HTTP or JMX (Java Management Extensions), which is a standard Java management technology. By using Actuator, developers can quickly see if an application is running correctly without the need to write custom monitoring code.

The most common way to use Actuator is through its "endpoints", which are specific web addresses that provide different types of information. For example,

[9] JUnit 5
[10] REST Assured
[11] SpringBoot Actuator

the health endpoint shows whether the application and its connected services, like databases, are functioning correctly, while the metrics endpoint displays detailed data on memory and CPU usage. Beyond the standard options, developers can also create their own custom endpoints or connect the data to external monitoring software to visualize how an application is performing over time.

Actuator can be enabled in a Spring Boot project by including the `spring-boot-starter-actuator` dependency. (Broadcom, Inc., 2026b)

4.5.8 Prometheus

Prometheus [12] is an open-source systems monitoring toolkit that was originally developed at SoundCloud and is now a project of the Cloud Native Computing Foundation. It is primarily used for collecting and storing multidimensional metrics as time-series data, meaning information is recorded with a timestamp and optional key-value pairs called labels. The system is designed for reliability and is capable of scraping data from instrumented jobs and web servers, storing it in a local time-series database, and triggering alerts based on predefined rules when specific thresholds are met. Through its powerful functional query language, PromQL, developers can aggregate and visualize performance data. (Prometheus Authors, 2026a, 2026b)

To collect and export Actuator metrics specifically for Prometheus, the `micrometer-registry-prometheus` dependency must be included in the classpath. (VMWare, Inc., n.d.) Access to the metrics is granted by including "prometheus" in the list of exposed web endpoints within the application's configuration properties. Once these components are in place, the metrics are automatically formatted for consumption and can be scraped by a Prometheus server. (Broadcom, Inc., 2026a)

4.5.9 Docker

Docker [13] is a platform used for developing and deploying applications. It is designed to separate software from the underlying infrastructure, allowing for faster delivery and consistent environments.

Docker's capabilities are centered around the use of containers, which are lightweight and isolated environments. Each container is packaged with all necessary dependencies required for an application to run, ensuring it operates independently of the host system. These workloads can be executed across different environments, such as local computers, data centers, or cloud providers, ensuring high portability. (Docker Inc., n.d.-b)

A Dockerfile is a text-based document containing a series of instructions for assembling a Docker image. Each command in this file results in the creation of a

[12] Prometheus
[13] Docker

layer in the image, making the final template efficient and fast to rebuild. These images serve as read-only blueprints from which runnable instances, or containers, are created. (Docker Inc., n.d.-c)

Docker Compose is a tool used to define and manage applications consisting of multiple containers. A single configuration file is used to specify the services, networks, and volumes required for the entire application stack. The lifecycle of complex applications can be managed with this tool, enabling all associated services to be started, stopped, and coordinated with a single command. (Docker Inc., n.d.-a)

4.5.10 k6

Grafana k6 [14] is an open-source performance testing tool designed to evaluate the reliability and performance of a system. It simulates various traffic patterns, such as constant load, sudden stress spikes, and long-term soak tests, to identify slow response times and system failures during development and continuous integration. Metrics are collected during execution and can be visualized through platforms like Grafana or exported to various data backends for detailed reporting. (Grafana Labs, n.d.-a)

k6 allows tests to be written in JavaScript, making it accessible and easy to integrate into existing codebases. Every k6 test follows a common structure. The main component is a function that contains the core logic of the test. This function should be the default export of the JavaScript file. It is executed concurrently for each Virtual User (VU), which act as independent execution threads to repeatedly apply the test logic. The tests can be enhanced using k6's lifecycle functions, such as a setup function, which is executed only once and may be utilized to insert seed data into the system. The test execution can be configured using an "options" object, where VUs, test duration and performance thresholds can be set. (Grafana Labs, n.d.-b)

[14] Grafana k6

Chapter 5

Implementation

5.1 Contract Test Implementation

The contract tests are implemented in a separate maven module called `test-suite`[1]. The test classes use the `JUnit 5` testing framework and `REST Assured` to send and assert HTTP requests. A basic test might look like this:

```
@Test
void getLectureDetails_shouldReturn200_returnTwoDates() {
    // First, create seed data
    var lectureSeedData = createLectureSeedData();

    RestAssured.given()
            .when()
            .get("/lectures/{lectureId}",
                lectureSeedData.lectureId())
            .then()
            .statusCode(200)
            .body("data.dates", hasSize(2));
}
```

Listing 5.1: Contract test example; adapted from `test-suite/src/test/karsch.lukas.lectures.AbstractLecturesE2ETest`

All contract tests follow a consistent pattern. First, they are annotated with `@DisplayName` to provide a descriptive, human-readable name. The test method itself is precisely named after the behavior it asserts. In the example above, the test verifies that the response status code is 200 and that the response body contains a field called `dates` consisting of an array of size two.

[1] `test-suite/src/test/java/karsch.lukas`

Before making these assertions, each test creates "seed data". Seed data is prerequisite data that must exist on the system under test for the execution to be valid. For instance, a professor, a course, and a lecture must be created before the endpoint to `GET` that specific lecture can be tested. Tests that assert invariants, such as the business rule preventing lecture from having overlapping timeslots, typically set the system time via a Spring Boot Actuator endpoint first.

Once the prerequisites are met, the request is executed and assertions are made using REST Assured. The `given()` block sets up the request requirements like headers, parameters, or body content; the `when()` block defines the action, such as the HTTP method (GET, POST) and the endpoint URL. Finally, the `then()` block is used to verify the response, allowing the developer to assert status codes and validate the data returned in the response body.

The test classes in `test-suite` are all `abstract`, meaning they can not be run directly. Instead, they are intended to be subclassed by the modules implementing the concrete applications (`impl-crud` & `impl-es-cqrs`). The subclasses must implement a set of abstract methods which are implementation specific, for example a method to reset the database in between each test, a method to set the application's time and methods to create seed data for tests.

Necessary infrastructure is spun up by the subclasses using Testcontainers. Testcontainers is a way to declare infrastructure dependencies as code and is an open-source library available for many programming languages. (Testcontainers, n.d.)

```
@TestConfiguration
public class PostgresTestcontainerConfiguration {
    @Bean
    @ServiceConnection
    @RestartScope
    PostgreSQLContainer<?> postgreSQLContainer() {
        return new PostgreSQLContainer<>(
                DockerImageName.parse("postgres:latest"));
    }
}
```

Listing 5.2: `impl-crud/src/test/karsch.lukas.PostgresTestcontainerConfiguration`

The above code snippet starts a PostgreSQL container using the latest available image. `@ServiceConnection` makes sure the Spring application can connect to the container. This configuration can then be imported into the test class:

```
@SpringBootTest
@Import(PostgresTestcontainerConfiguration.class)
```

```
public class CrudLecturesE2ETest extends AbstractLecturesE2ETest { }
```

Listing 5.3: `impl-crud/src/test/karsch.lukas.e2e.lectures.CrudLecturesE2ETest`

5.2 CRUD implementation

This section presents the relevant aspects of the CRUD implementation[2], mainly focusing on relational modeling using JPA and the audit log implementation.

5.2.1 Relational Modeling

The CRUD implementation uses a normalized database in the Third Normal Form.

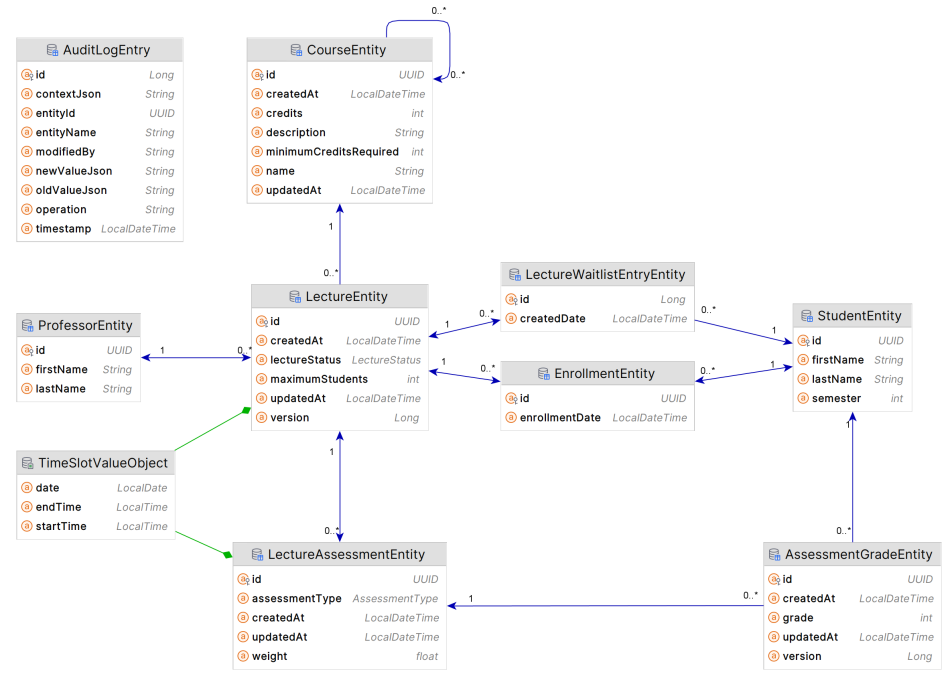

Figure 5.1: Entity Relationship Diagram for the CRUD App

Figure 5.1 shows the Entity Relationship Diagram for the CRUD app. It includes nine entities and a value object for the app's relational database schema. Each box corresponds to an entity or value object, with the bold text being the name. Below the table's name, all attributes of the entity are listed with their type and name.

[2] `impl-crud/src/main/java/karsch.lukas`

Arrows represent an association. The numbers at the end of the arrows convey the multiplicity. An arrow pointing in only one direction stands for a unidirectional association, while an arrow pointing in both directions conveys a bidirectional association. For example, an arrow pointing between entity `A` and entity `B` like so: `1` $\longleftrightarrow$ `0..1` shows that one `A` can be associated with any number of `B`'s, and a `B` is always associated with exactly one `A`.

Arrows with a filled diamond represent a composition. Compositions are used when an entity has a reference to a value object. This value object has no identity and is directly embedded into the entity. The only value object in figure 5.1 is the `TimeSlotValueObject`.

In the app's ER diagram, the `LectureEntity` serves as core of the schema, having several key associations. The 0..* $\longrightarrow$ 1 association to `CourseEntity` shows that many lectures can be created from a course and a lecture is always associated with a course. The 0..* $\longrightarrow$ 1 association to `ProfessorEntity` shows that a professor can hold many lectures (or none), and that a lecture is always associated with a professor. From the lecture's side, these relationships are called "Many to One" relationships.

`LectureEntity` also has "One to Many" relationships to `LectureWaitlistEntryEntity`, `EnrollmentEntity` and `LectureAssessmentEntity`. `LectureWaitlistEntryEntity` is a table which stores students who are waitlisted for a lecture. It is effectively a join table (with one extra column to track when the student was waitlisted) and represents a Many to Many relationship between lectures and students. The same applies to `EnrollmentEntity` which is a table storing which students are enrolled to which lecture. `LectureAssessmentEntity` represents the fact that a lecture can have many assessments (which may be an exam, a paper or a project). Each assessment in turn has many `AssessmentGradeEntity`s associated with it. This table stores which student scored which grade on an assessment.

The `AuditLogEntry` is also visible on the ER diagram, however it has no relationships. This table and the entire audit log implementation will be laid out in the following section.

These entities are implemented using SpringBoot's JPA integration. For example, an entity with a "One to Many" relationship can be implemented like this:

```
@Entity
class LectureEntity {
    @Id
    private UUID id;

    @OneToMany(fetch=FetchType.LAZY)
    private List<EnrollmentEntity> enrollments;
```

```
}
```

Listing 5.4: Simple JPA entity with a "One to Many" relationship

The `@Entity` annotation informs JPA that the class should be mapped to a database table. If the schema generation feature is enabled, JPA automatically creates a table structure that mirrors the class definition. In production environments where this feature is typically disabled, developers must provide SQL scripts to manually define the expected structure. This is commonly achieved either by including a basic initialization script or by utilizing dedicated database migration tools such as Flyway or Liquibase to manage versioned schema changes.

Each entity must include a field annotated with `@Id`, which serves as the unique primary key for the corresponding database record.

The `@OneToMany` annotation defines a relational link between two entities. While the collection is accessed in Java as a standard list via `lecture.getEnrollments()`, JPA manages this behind the scenes using a foreign key relationship. The `fetch` parameter determines when this data is retrieved: `LAZY` loading defers the database query until the collection is explicitly accessed in the code, whereas `EAGER` loading fetches the related entities immediately alongside the parent object.

5.2.2 Audit Log implementation

There are several strategies to implement an audit log, each with its own trade-offs:

1. **Manual Logging**: Developers explicitly call a logging service in every service method that modifies data. While simple, this can lead to code duplication and is prone to human error, such as developers forgetting to add a log statement. A code example might look like this:

```
public void updatePhoneNumber(User user, int newNumber) {
    logChange(Date.now(), user, user.getPhoneNumber(),
        newNumber, "UserRequestedNumberChange");
    user.setPhoneNumber(newNumber);
}

void logChange(
    Date date, User user, Object oldValue, Object newValue,
        String context
) {
    LogEntry logEntry = new LogEntry(date, user, oldValue,
        newValue);
    logRepository.persist(logEntry);
}
```

Listing 5.5: Code example for manual audit logging

2. **Database Triggers or Stored Procedures** can capture changes automatically and directly on the database. This guarantees that no change is missed, even if made outside the application. Ingram (2009, p. 515) mentions that database triggers run on a "per-record" basis, meaning the logic is run for each changed record individually. This may lead to degraded performance during batch operations, which is why stored procedures should be preferred over triggers for auditing concerns. It is also worth noting that this approach ties the auditing logic to a specific database, making it less portable.

3. **Hibernate Envers** is an auditing solution for JPA-based applications which automatically versions entities by using the concept of revisions. Envers creates an auditing table for each entity which stores historical data, whenever a transaction is committed. (Hibernate, n.d.)

4. **JPA Entity Listeners**: JPA's lifecycle events (`@PrePersist`, `@PreUpdate`, etc.) can be used to intercept changes. This approach is database-independent and keeps the logic within the Java application, allowing access to application internals like beans and Spring's security context. In full-grade applications built using Spring Security, the security context lets developers access the current user, making it possible to attach them to the new audit log entry. (needs reference)

5.2.3 Chosen Implementation: JPA Entity Listener

The CRUD implementation of the application utilizes approach 4, JPA entity listeners, which offer a good balance between automation and flexibility. This approach ensures that every change to an entity is captured without polluting the service layer with logging calls, while still allowing the application to enrich the log with application-level context. The current user is automatically added to the log entry by the entity listener, while service methods can attach additional context to an entity before it is persisted, if desired.

Data Model

The audit log is stored in a single database table, represented by the `AuditLogEntry` entity. This structure allows for easy querying of all system changes in chronological order. Table 5.1 lists the fields contained by `AuditLogEntry`. This data model is adapted from Fowler (2004).

Field	Type, Possible Values	Explanation
entityName	String	Identifies the type of the changed object.
entityId	UUID	Unique identifier of the specific changed object.
timestamp	LocalDateTime	The date and time when the change occurred.
operation	CREATE, UPDATE, DELETE	The type of action performed on the entity.
modifiedBy	String	The user or system process that initiated the change.
oldValueJson	TEXT (JSON String)	The serialized state of the entity before the change.
newValueJson	TEXT (JSON String)	The serialized state of the entity after the change.
contextJson	TEXT (JSON String)	Captures additional business intent or metadata.

Table 5.1: Audit Log Entry Structure

AuditableEntity and AuditEntityListener

To enable auditing, entities extend an abstract base class, `AuditableEntity` [3]. This class marks the entity with the `@EntityListeners(AuditEntityListener.class)` annotation and provides a transient field, `snapshotJson`, used to store the state of the entity when it is loaded from the database.

The core logic resides in `AuditEntityListener`[4]. It hooks into the JPA lifecycle:

- `@PostLoad`: Immediately after an entity is fetched from the database, the listener serializes it to JSON and stores the serialized JSON string in the `snapshotJson` field. This serves as the "old value" for any subsequent updates.
- `@PrePersist`: Before a newly created entity is saved, a log entry is created with the operation `CREATE`. The `newValueJson` is the current state; `oldValueJson` is null.
- `@PreUpdate`: Before an existing entity is updated, the listener compares the current state with the `snapshotJson`. It creates an `UPDATE` entry using `snapshotJson` as the `oldValue` and the current state as the `newValue`.

[3] `impl-crud/src/main/java/karsch.lukas.audit.AuditableEntity`
[4] `impl-crud/src/main/java/karsch.lukas.audit.AuditEntityListener`

- `@PreRemove`: Before deletion, a `DELETE` entry is created, preserving the last known state. `snapshotJson` is saved as the `oldValue`, `newValue` is `null`

Serialization and Relationships

A major challenge in serializing JPA entities to JSON for an audit log is handling relationships. A naive JSON serialization would follow every relationship of the affected entity (e.g., `Lecture` → `Course`), potentially serializing huge entity graphs or even causing a `StackOverflowError` due to cyclic references. One example for such a problematic reference would be the self-reference of `CourseEntity`, which each course having a "One to Many" relationship to prerequisite courses.

A custom Jackson module, `IdSerializationModule`[5], was implemented to deal with the problem of relationship serialization. This module overrides the default serialization for related entities. Instead of serializing the full nested object, it only writes the related object's ID. This results in a flat, lightweight JSON structure that is readable, keeps all references reconstructible and is safe to store.

Capturing Business Context

One downside of standard automated auditing is that it captures *what* changed (e.g., lecture lifecycle changed from `IN_PROGRESS` to `FINISHED`) but not necessarily *why* (e.g., Lecture cancelled by professor due to illness). To make the CRUD application's audit log compliant to the requirement of traceability, it supports an `AuditContext`. Services can attach metadata to the current thread/transaction, which the `AuditEntityListener` retrieves and stores in the `contextJson` field.

5.3 ES/CQRS implementation

5.3.1 Architecture Overview

The architecture of the `impl-es-cqrs` application [6] differs from the traditional layered architecture seen in the `impl-crud` application. While the CRUD implementation also has some vertical slicing, the ES-CQRS implementation is much more explicit about it. The code is organized into "features", each representing a vertical slice of the application's functionality (e.g., `course`, `enrollment`, `lectures`). Each feature is self-contained and includes its own command handlers, event sourcing handlers, query handlers, and its own web controller, if needed.

A "feature slice" architecture is descriptive and able to communicate the features of a project at a glance. As clean architecture is not in the scope of this thesis, the separation into features with clear naming conventions for command and query

[5] `impl-crud/src/main/java/karsch.lukas.audit.IdSerializationModule`
[6] `impl-es-cqrs/src/main/java/karsch.lukas`

components is sufficient, however introducing completely separate modules for the command and read sides would have increased the project structure's readability even more by clearly showing how command and read side have no access to each other.

5.3.2 The API Layer

The `api` package in each feature slice is shared between web controllers, command side and read side, containing the public interface of the application. It defines the Commands, Events, and Queries that are dispatched and handled by the `impl-es-cqrs` application. Keeping the public API in a separate package ensures that the internal implementation details of the `impl-es-cqrs` application are not exposed to its clients.

5.3.3 Command Side

The command side is responsible for handling state changes in the application. It is implemented using Axon's Aggregates, Command Handlers, and Sagas. This section goes in detail about the implementation aspects, using the courses feature as an example.

Aggregates and Set-Based Validation

Aggregates are the core components of the command side. They represent a consistency boundary for state changes. In this implementation, an example of an aggregate is the `CourseAggregate` [7]. It handles the `CreateCourseCommand`, validates it, and if successful, emits a `CourseCreatedEvent`.

A core aspect of the validation within the `CourseAggregate` is set-based validation. Before creating a course, the system must verify that all the specified prerequisite courses actually exist. This is handled by the `ICourseValidator`[8], which is injected into the aggregate's command handler. The validator uses an SQL lookup table, implemented using a JPA entity and repository, that is maintained by the `CourseLookupProjector`. This ensures that the command is validated against a consistent view of the system's state. It is important to note that while projectors typically belong to the read side, "lookup projectors" in this application belong to the command side. They are independent to the read side and maintain their own view of the system. This view is not eventually consistent, but strongly consistent, making it suitable for command validation. A strongly consistent lookup

[7] `impl-es-cqrs/src/main/java/karsch.lukas.features.course.commands.CourseAggregate`

[8] `impl-es-cqrs/src/main/java/karsch.lukas.features.course.commands.ICourseValidator`

table is achieved through the use of subscribing event listeners, which are executed immediately after an event has been applied.

External Command Handlers

Not all commands can be handled by a single aggregate. For instance, assigning a grade to a student for a specific lecture involves the `EnrollmentAggregate` and the `LectureAggregate`. In such cases, a dedicated command handler, `EnrollmentCommandHandler`, is used. This handler coordinates the interaction between the aggregates. It loads the `EnrollmentAggregate` from the event sourcing repository, validates the command (e.g., checking if the professor is allowed to assign a grade for the lecture), and then executes the command on the aggregate.

Sagas for Process Management

Sagas are used to manage long-running business processes that span multiple aggregates. The `AwardCreditsSaga` is a prime example. It is initiated when an `EnrollmentCreatedEvent` occurs. The saga then waits for a `LectureLifecycleAdvancedEvent` with the status `FINISHED`. Once this event is received, the saga sends an `AwardCreditsCommand` to the `EnrollmentAggregate`. The saga ends when it receives a `CreditsAwardedEvent`. This ensures that credits are only awarded after a lecture is finished and all assessments have been graded. It is interesting to note that while the CRUD application calculates awarded credits based on the current state of a lecture, in the ES-CQRS implementation, the fact that credits are awarded after finishing a lecture is explicit. Even when changing the Saga later on, credits which have already been awarded will not be revoked, unless additional, explicit logic is implemented (e.g. by applying a `CreditsRevokedEvent`).

5.3.4 Read Side

The read side listens to events asynchronously and builds read models, called "projections", which are views of the system. A component that listens for events and maintains projections is called a "projector". Projections are designed to answer specific questions about the system: each projector saves exactly the necessary information. This is achieved by using denormalized data models, a contrast to typical CRUD systems that follow normalization rules.

When the system is queried, the queries are routed to the read side. The read side can efficiently fetch data from the projections, usually without `JOINs`. This makes reads fast. It is important to keep in mind that projections are built asynchronously, meaning they are eventually consistent and may not always reflect the latest changes applied by the command side.

In the context of the ES-CQRS implementation, a good example of a projector that stores denormalized data for efficient querying is the `LectureProjector`. It demonstrates the fact that each projector maintains its own view of the system. Projectors must not query the system using Axon's `QueryGateway` to get access to any data needed for the projection. One reason for that is the fact that when *rebuilding* projections, a common use case in event sourcing, the projectors should be able to run in parallel. If projectors depend on each other, this can result in one projection attempting to query data from another projection that is not yet up to date. This is why the `LectureProjector` not only maintains a view of lectures, but also of courses, professors and students, which are then used when building the lecture's projection.

The projector also illustrates how the projection's database entities are designed: they are built in the same way as the DTO which is returned from the query handler. Arrays and associated objects are not stored via foreign keys but are instead serialized to JSON. This allows the retrieval of all the necessary data to respond to a query with a simple `SELECT` statement. The same concepts apply to all other projectors in the ES-CQRS implementation.

5.3.5 Synchronous Responses with Subscription Queries

A common challenge in CQRS and event-driven architectures is providing synchronous feedback to users. For example, when a student enrolls in a lecture, they expect an immediate response indicating whether they were successfully enrolled or placed on a waitlist. However, commands are usually handled asynchronously. In CQRS, commands are also not intended to return data.

To solve this, the `LecturesController` uses Axon's subscription queries. When an enrollment request is received, it sends the `EnrollStudentCommand` and simultaneously opens a subscription query (`EnrollmentStatusQuery`). This query waits for an `EnrollmentStatusUpdate` event. The read-side projector responsible for processing enrollments publishes this update after processing the respective `StudentEnrolledEvent` or `StudentWaitlistedEvent`. The controller blocks for a short period, waiting for this update to be published, and then returns the result to the user. This approach makes the user interface synchronous, while not contradicting with the asynchronous nature of CQRS systems, as the command handling process is unchanged. While this approach provides the desired synchronous user experience, it has the downside of coupling the client to the event processing flow. In a typical scenario, one might use WebSockets or other client-side notification mechanisms to inform the user about the result of their action. However, for the context of this thesis, where the focus is on the backend implementation and contract testing, this solution is a pragmatic compromise.

5.3.6 Encapsulation and API Boundaries

To enforce the separation of concerns and maintain a clean architecture, the internal components of the command and read sides are package-private. For example, the `CourseAggregate` and `CourseProjector` are not accessible from outside their respective feature packages. The public API of the application is exposed through the controllers, which only interact with the `CommandGateway` and `QueryGateway`. This ensures that all interactions with the system go through the proper channels and that internal implementations can be changed without affecting the clients.

5.3.7 Tracing Request Flow

This section illustrates the flow of commands and queries through the system. Axon's `CommandGateway` and `QueryGateway` are used in controllers to decouple them from the internals of the application. The gateways create location transparency: a controller does not need to know where its commands and queries are being routed to.

Command Request: CreateCourseCommand

Figure 5.2 illustrates the flow of a command through the system using the example of the `POST /courses` endpoint. Upon receiving a request, the controller constructs a `CreateCourseCommand` containing the request data and dispatches it through the `CommandGateway`. This gateway is responsible for routing the command to the appropriate destination, which in this case is the constructor of the `CourseAggregate`. This constructor is annotated with `@CommandHandler`. The command handler verifies that the command is allowed to be executed by performing validation logic. When creating courses, it has to be made sure that all prerequisite courses actually exist. This check is done using set-based validation. If the validation is successful, the aggregate triggers a state change by applying a `CourseCreatedEvent` via the `AggregateLifecycle.apply()` method. This action notifies the system of the change and persists the event by recording it in the event store.

After being applied, Axon routes the event to all subscribed handlers. The `CourseAggregate`'s `@EventSourcingHandler` is executed, changing the aggregate's internal state. What is worth noting here is that in the case of `CourseAggregate`, only the `id` of the course is set as other properties of the event, like name or description of the newly created course, are not relevant to the command side. Any read-side projectors with `@EventHandlers` for the `CourseCreatedEvent` are also executed after the event is applied.

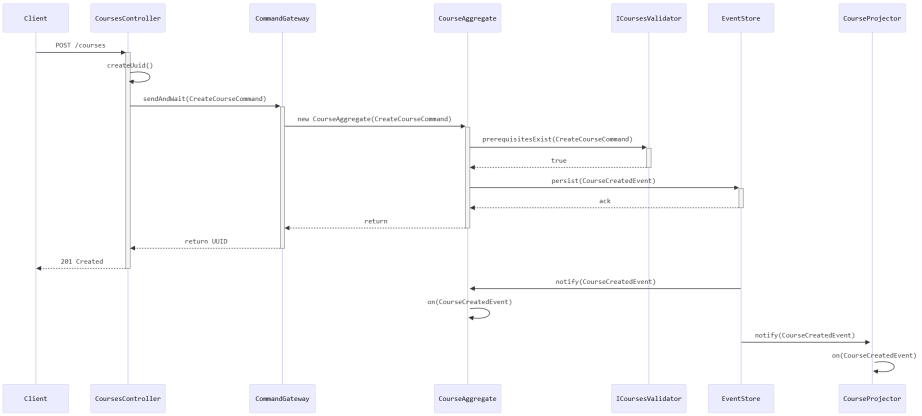

Figure 5.2: Sequence Diagram: Command Flow inside the ES-CQRS application

Query Request: FindAllCoursesQuery

Figure 5.3 illustrates the flow of a query through the application using the `GET /courses` request as an example. The request is received by `CoursesController`. It creates a `FindAllCoursesQuery` instance and sends it to Axon's `QueryGateway`, which routes the query to the appropriate `@QueryHandler` method responsible for `FindAllCoursesQuery`. The query handler method then accesses its JPA repository to get all courses, maps them to a list of `CourseDTOs` and returns this list. The `QueryGateway` hands this result over to the web controller which reads the data and sends it back to the client.

5.4 Infrastructure

5.5 Performance Tests

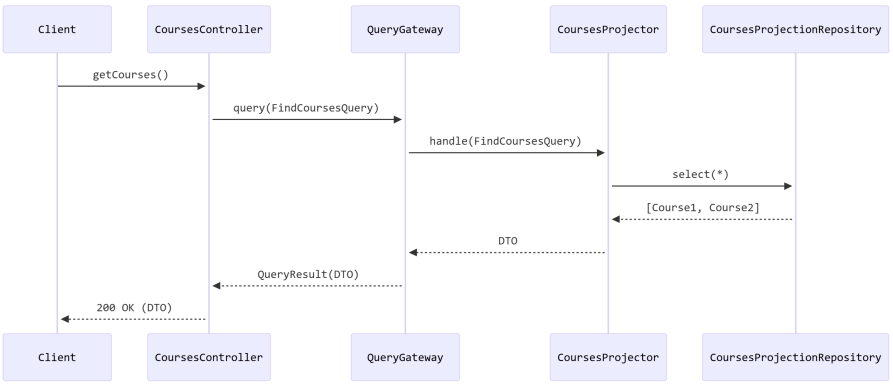

Figure 5.3: Sequence Diagram: Query Flow inside the ES-CQRS application

Chapter 6

Results

		CRUD (ms)		ES-CQRS (ms)			
Metric	**Users**	**Mean**	**CI ±**	**Mean**	**CI ±**	**Speedup**	**Significance**
Avg	500	840.33	0.00	1.47	0.00	573x	***
Median	500	913.11	0.00	1.25	0.00	728x	***
P95	500	2146.04	0.01	2.85	0.00	754x	***
P99	500	2922.25	0.03	5.08	0.00	575x	***

Table 6.1: Statistical comparison of latency for GET /lectures (500 Virtual Users) over 30 iterations. A Significance of *** indicates a p-value ≤ 0.001

Chapter 7

Discussion

7.1 Analysis of results

7.2 Conclusion & Further work

Finally, I'm done!

Bibliography

Apache Groovy project. (n.d.-a). The Apache Groovy programming language - Processing XML. Retrieved January 8, 2026, from https://groovy-lang.org/processing-xml.html#_gpath

Apache Groovy project. (n.d.-b). The Apache Groovy™ programming language. Retrieved January 8, 2026, from https://groovy-lang.org/

ATIS Committee. (2013, March). ATIS Telecom Glossary - audit trail. Retrieved January 7, 2026, from https://web.archive.org/web/20130313232104/http://www.atis.org/glossary/definition.aspx?id=5572

Axoniq. (2025a). Axon Server - Event Store & Message Delivery System. Retrieved January 12, 2026, from https://www.axoniq.io/server

Axoniq. (2025b). Command Dispatchers. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/command-dispatchers/

Axoniq. (2025c). Command Handlers. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/command-handlers

Axoniq. (2025d). Event Bus & Event Store. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/events/infrastructure/

Axoniq. (2025e). Event Handlers. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/events/event-handlers/

Axoniq. (2025f). Infrastructure. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/infrastructure/

Axoniq. (2025g). Introduction (5.0). Retrieved January 12, 2026, from https://docs.axoniq.io/axon-framework-reference/5.0/

Axoniq. (2025h). Introduction (v2025.2). Retrieved January 12, 2026, from https://docs.axoniq.io/axon-server-reference/v2025.2/

Axoniq. (2025i). Messaging Concepts (4.12). Retrieved January 12, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/messaging-concepts/

Axoniq. (2025j). Multi-Entity Aggregates. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/modeling/multi-entity-aggregates/

Axoniq. (2025k). Query Dispatchers. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/queries/query-dispatchers/

Axoniq. (2025l). Saga Associations. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/sagas/associations/

Axoniq. (2025m). Saga Implementation. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/sagas/implementation/#_injecting_resources

Axoniq. (2025n). Streaming Event Processor. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/events/event-processors/streaming/

Axoniq. (2025o). Subscribing Event Processor. Retrieved January 23, 2026, from https://docs.axoniq.io/axon-framework-reference/4.12/events/event-processors/subscribing/

Bauer, C. (2016). *Java persistence with Hibernate* (Second edition). Manning Publications.

Bernstein, P. A., & Newcomer, E. (2009). *Principles of transaction processing* (2nd edition). Morgan Kaufmann Publishers.

Braun, S., Deßloch, S., Wolff, E., Elberzhager, F., & Jedlitschka, A. (2021). Tackling Consistency-related Design Challenges of Distributed Data-Intensive Systems - An Action Research Study [arXiv:2108.03758 [cs]]. *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–11. https://doi.org/10.1145/3475716.3475771

Broadcom, Inc. (2026a). Metrics :: Spring Boot. Retrieved January 19, 2026, from https://docs.spring.io/spring-boot/reference/actuator/metrics.html#actuator.metrics.export.prometheus

Broadcom, Inc. (2026b). Production-ready Features :: Spring Boot. Retrieved January 19, 2026, from https://docs.spring.io/spring-boot/reference/actuator/index.html

Broadcom, Inc. (2026c). Why Spring. Retrieved January 12, 2026, from https://spring.io/why-spring

Committee on National Security Systems. (2010, April). National Information Assurance Glossary. Retrieved January 7, 2026, from https://web.archive.org/web/20120227163121/http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf

Deinum, M., Rubio, D., & Long, J. (2023). *Spring 6 Recipes: A Problem-Solution Approach to Spring Framework*. Apress. https://doi.org/10.1007/978-1-4842-8649-4

Docker Inc. (n.d.-a). What is Docker Compose? Retrieved January 19, 2026, from https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-docker-compose/

Docker Inc. (n.d.-b). What is Docker? Retrieved January 19, 2026, from https://docs.docker.com/get-started/docker-overview/

Docker Inc. (n.d.-c). Writing a Dockerfile. Retrieved January 19, 2026, from https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/

Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley.

FasterXML. (2025, October). Jackson Project Home @github [original-date: 2011-10-19T05:28:40Z]. Retrieved January 19, 2026, from https://github.com/FasterXML/jackson

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* [Doctoral dissertation, University of California]. Retrieved January 8, 2026, from https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf

Fowler, M. (2003, November). Anemic Domain Model. Retrieved December 27, 2025, from https://martinfowler.com/bliki/AnemicDomainModel.html

Fowler, M. (2004, April). Audit Log. Retrieved November 13, 2025, from https://martinfowler.com/eaaDev/AuditLog.html

Fowler, M. (2005, December). Event Sourcing. Retrieved November 13, 2025, from https://martinfowler.com/eaaDev/EventSourcing.html

Grafana Labs. (n.d.-a). Grafana k6. Retrieved January 19, 2026, from https://grafana.com/docs/k6/latest/

Grafana Labs. (n.d.-b). Write your first test. Retrieved January 19, 2026, from https://grafana.com/docs/k6/latest/get-started/write-your-first-test/

Gray, J., Helland, P., O' Neil, P., & Sasha, D. (1996). The dangers of replication and a solution. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. https://doi.org/10.1145/233269.233330

Hibernate. (n.d.). Envers - Hibernate ORM. Retrieved January 20, 2026, from https://hibernate.org/orm/envers/

Ingram, D. (2009). *Design – Build – Run: Applied Practices and Principles for Production-Ready Software Development*. Retrieved November 14, 2025, from https://www.oreilly.com/library/view/design-build/9780470257630/

Jacobs, I., & Walsh, N. (2004, December). Architecture of the World Wide Web, Volume One. Retrieved December 27, 2025, from https://www.w3.org/TR/webarch/

JetBrains. (2023). Java Programming - The State of Developer Ecosystem in 2023 Infographic. Retrieved January 12, 2026, from https://www.jetbrains.com/lp/devecosystem-2023

Johan Haleby. (n.d.). REST Assured Documentation. Retrieved January 8, 2026, from https://github.com/rest-assured/rest-assured/wiki/Usage

Joint Task Force Interagency Working Group. (2020, September). *Security and Privacy Controls for Information Systems and Organizations* (tech. rep.) (Edition: Revision 5). National Institute of Standards & Technology. https://doi.org/10.6028/NIST.SP.800-53r5

JUnit User Guide. (n.d.). Retrieved January 8, 2026, from https://docs.junit.org/5.14.2/overview.html

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly.

Malyi, R., & Serdyuk, P. (2024). Developing a Performance Evaluation Benchmark for Event Sourcing Databases. *Vìsnik Nacìonal'nogo unìversitetu "L'vìvs'ka polìtehnìka". Serìâ Ìnformacìjnì sistemi ta merežì*, *15*, 159–168. https://doi.org/10.23939/sisn2024.15.159

Martin, J. (1983). *Managing the data-base environment*. Prentice-Hall.

Meyer, B. (2006). STANDARD EIFFEL.

Michelson, B. (2006, February). *Event-Driven Architecture Overview* (tech. rep. No. 681). Patricia Seybold Group. Boston, MA. https://doi.org/10.1571/bda2-2-06cc

Oracle. (n.d.). Jackson JSON processor. Retrieved January 19, 2026, from https://docs.oracle.com/en/middleware/goldengate/core/23/ogglc/jackson-json-processor.html

PostGIS PSC. (2023). PostGIS. Retrieved January 12, 2026, from https://postgis.net/

PostgreSQL Global Development Group. (2026, January). PostgreSQL. Retrieved January 12, 2026, from https://www.postgresql.org/

Prometheus Authors. (2026a). Overview | Prometheus. Retrieved January 19, 2026, from https://prometheus.io/docs/introduction/overview/

Prometheus Authors. (2026b). Prometheus - Monitoring system & time series database. Retrieved January 19, 2026, from https://prometheus.io/

Richards, M. (2015). *Software Architecture Patterns*. O'Reilly. Retrieved January 8, 2026, from https://theswissbay.ch/pdf/Books/Computer%20science/O'Reilly/software-architecture-patterns.pdf

Testcontainers. (n.d.). Testcontainers. Retrieved January 8, 2026, from https://testcontainers.com/

The Internet Society. (1999). RFC 2616: HTTP/1.1. Retrieved December 27, 2025, from https://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf

Tilkov, S. (2007, December). A Brief Introduction to REST. Retrieved January 8, 2026, from https://www.infoq.com/articles/rest-introduction/

U.S. Securities and Exchange Commission. (2012, August). 17 CFR § 242.613 - Consolidated Audit Trail. Retrieved January 7, 2026, from https://www.law.cornell.edu/cfr/text/17/242.613

VMWare, Inc. (n.d.). Micrometer Prometheus :: Micrometer. Retrieved January 19, 2026, from https://docs.micrometer.io/micrometer/reference/implementations/prometheus

Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, *52*(1), 40–44. https://doi.org/10.1145/1435417.1435432

Walls, C. (2016). *Spring Boot in Action*. Manning Publications Co. LLC.

Young, G. (2010). CQRS Documents by Greg Young. Retrieved December 26, 2025, from https://cqrs.wordpress.com/wp-content/uploads/2010/11/cqrs_documents.pdf

Appendix A

Source Code

The full source code for this thesis, including both apps, performance tests and markdown notes, is available at: https://gitlab.mi.hdm-stuttgart.de/lk224/thesis