



Bachelor's Thesis in Computer Science and Media

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

Lukas Karsch

45259

Hochschule der Medien Stuttgart

Submitted on 2026/03/02

to obtain the degree of Bachelor of Science

Main Supervisor: Prof. Dr. Tobias Jordine

Secondary Supervisor: Felix Messner

Ehrenwörtliche Erklärung

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research question(s)	1
1.3	Goals and non goals	1
1.4	Structure of the paper	1
2	Basics	1
2.1	WWW, Web APIs, REST	1
2.2	Layered Architecture Foundations	2
2.3	Domain Driven Design	3
2.4	CRUD architecture	5
2.5	CQRS Architecture	5
2.6	(Eventual) Consistency	6
2.7	Event Sourcing and event-driven architectures	7
2.8	Traceability and auditing in IT systems	7
2.8.1	Audit Logs	7
2.8.2	Event Streams	8
2.8.3	Rebuilding state from an audit log and an event stream	8
2.9	Scalability of systems	8
3	Related Work	8
4	Proposed Method	8
4.1	Project requirements	9
4.1.1	Entities	9
4.1.2	Business rules	9
4.1.3	Contract Tests	10
4.2	Performance	10
4.3	Scalability or flexibility (TODO)	10
4.4	Traceability	10
4.5	Tech Stack	10
4.5.1	SpringBoot	10
4.5.2	JPA	11
4.5.3	PostgreSQL	12
4.5.4	Jackson	12
4.5.5	Axon	12
4.5.6	Testing	13
4.5.7	SpringBoot Actuator	13
4.5.8	Prometheus	14

4.5.9	Docker	15
4.5.10	k6	15
5	Implementation	15
5.1	Contract Test Implementation	15
5.2	CRUD implementation	17
5.2.1	Relational Modeling	17
5.2.2	Audit Log implementation	19
5.3	ES/CQRS implementation	23
5.4	Infrastructure	23
5.5	Performance Tests	23
6	Results	23
7	Discussion	23
7.1	Analysis of results	23
7.2	Conclusion & Further work	23
A	Source Code	29

List of Figures

1	Entity Relationship Diagram for the CRUD App	17
---	--	----

1 Introduction

1.1 Motivation

1.2 Research question(s)

1.3 Goals and non goals

1.4 Structure of the paper

2 Basics

2.1 WWW, Web APIs, REST

The World Wide Web (WWW) is a connected information network used to exchange data. Resources 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 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. (Fowler 2005) 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. (Fowler 2017; Lima et al. 2021)

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) 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)

Because every action taken on the system is stored, a number of facilities can be built on top of the event log: Temporal queries can be made, which determine the exact state of the application at any point in time. The event log acts as an immutable audit trail, making Event Sourcing architectures highly valuable for systems like accounting applications. (Fowler 2005)

2.8 Traceability and auditing in IT systems

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)

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 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)

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

2.8.3 Rebuilding state from an audit log and an event stream

2.9 Scalability of systems

3 Related Work

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 is an open-source, opinionated framework for developing enterprise Java applications. It is based on Spring Framework, which is a platform aiming to make Java development "quicker, easier, and safer for everybody" (*Why Spring* 2026). Spring provides an Inversion of Control (IoC) container which can be used for dependency injection. (Deinum, Rubio, & Long 2023,

Chapter 1) It offers support for several programming paradigms: reactive, event-driven, microservices and serverless. (*Why Spring* 2026)

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. (*Java Programming - The State of Developer Ecosystem in 2023 Infographic* 2026)

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 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), 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 bridge 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* is used as implementation of the JPA stan-

dard. (Bauer 2016, Chapter 1)

4.5.3 PostgreSQL

PostgreSQL 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* 2026) As of January 2026, PostgreSQL 18 is the latest version. (PostgreSQL Global Development Group 2026)

4.5.4 Jackson

Jackson 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 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. (*Jackson JSON processor* 2026; FasterXML 2025)

4.5.5 Axon

Axon Framework 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 2025b; Axoniq 2025d)

Axon Server 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 2025c; Axoniq 2025a)

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

JUnit 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* 2026)

REST Assured 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. (*REST Assured Documentation* 2026)

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)
    );
```

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 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 on the application's status, gather performance data, and track HTTP requests. Developers can interact with these features using either web-based HTTP links or JMX, 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, the health endpoint shows if the application and its connected services, like databases, are working, while the metrics endpoint displays detailed data on memory and CPU usage. Actuator also integrates with cloud platforms like Kubernetes to provide status updates for automated systems. Beyond the standard options, you can also create your own custom endpoints or connect the data to external monitoring software to visualize how your application is performing over time.

Actuator can be enabled in a Spring Boot project by including the **spring-boot-starter-actuator** dependency. (*Production-ready Features :: Spring Boot* 2026)

4.5.8 Prometheus

Prometheus is an open-source systems monitoring toolkit originally built at SoundCloud that is now a project under the Cloud Native Computing Foundation. It is primarily used for collecting and storing multi-dimensional 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 called "PromQL", developers can aggregate and visualize performance data. (*Prometheus - Monitoring system & time series database* 2026; *Overview / Prometheus* 2026)

To collect and export Actuator metrics specifically for Prometheus, the **micrometer-registry-prometheus** dependency must be included in the classpath. (*Micrometer Prometheus :: Micrometer* 2026) 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, metrics are automatically formatted for consumption and can be scraped by a Prometheus server. (*Metrics :: Spring Boot* 2026)

4.5.9 Docker

4.5.10 k6

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));
}
2
```

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.

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 require-

¹test-suite/src/test/java/karsch.lukas

²Adapted from test-suite/src/test/karsch.lukas.lectures.
AbstractLecturesE2ETest

ments 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* 2026)

```
@TestConfiguration
public class PostgresTestcontainerConfiguration {
    @Bean
    @ServiceConnection
    @RestartScope
    PostgreSQLContainer<?> postgresSQLContainer() {
        return new PostgreSQLContainer<>(
            DockerImageName.parse("postgres:latest"));
    }
}
3
```

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 { }
4
```

³`impl-crud/src/test/karsch.lukas.PostgresTestcontainerConfiguration`

⁴`impl-crud/src/test/karsch.lukas.e2e.lectures.CrudLecturesE2ETest`

5.2 CRUD implementation

This chapter presents the relevant aspects of the CRUD implementation, 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.

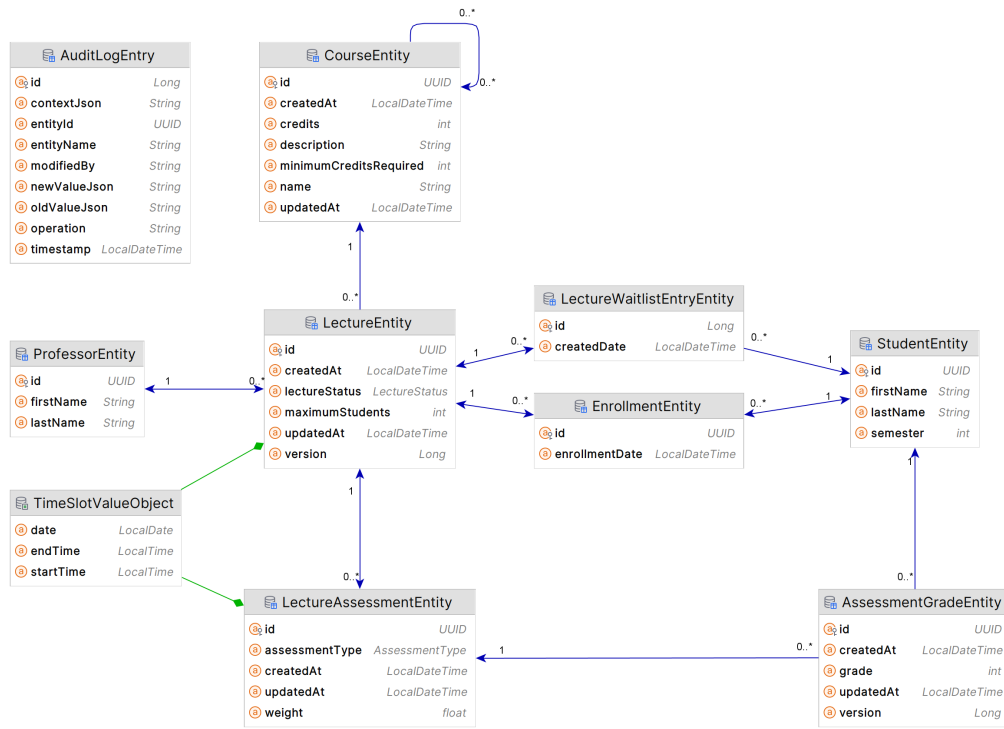


Figure 1: Entity Relationship Diagram for the CRUD App

Figure 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.

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 1 is the `TimeSlotValueObject`.

In the app's ER diagram, the `LectureEntity` serves as core of the schema, having several key associations. The $0..* \rightarrow 1$ association to `CourseEntity` shows that many lectures can be created from a course and a lecture is always associated with a course. The $0..* \rightarrow 1$ 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;
}
```

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);
}
```


2. **Database Triggers:** Database-level triggers can capture changes automatically. This offers high performance and guarantees that no change is missed, even if made outside the application. However, this approach ties the logic to a specific database and makes business logic harder to maintain and test, as it resides outside the codebase (needs reference).
3. **Hibernate Envers:** A popular solution for JPA-based applications. It automatically versions entities. However, Envers typically creates a shadow table for every entity, which can be overkill if only specific high-level changes need to be tracked, and querying the history can be complex. (needs reference)
4. **JPA Entity Listeners:** Using the JPA specification's lifecycle events (`@PrePersist`, `@PreUpdate`, etc.) to intercept changes. This approach is database-independent and keeps the logic within the Java application, allowing access to Spring's security context. The security context makes it easy to access the current user, making it possible to attach them to the new audit log entry. (needs reference)

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 1 lists the fields contained by `AuditLogEntry`. This data model is adapted from Fowler (2004).

AuditableEntity and AuditEntityListener

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 1: Audit Log Entry Structure

To enable auditing, entities extend an abstract base class, `AuditableEntity`⁵. 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`⁶. 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`

⁵`impl-crud/src/main/karsch.lukas.audit.AuditableEntity`

⁶`impl-crud/src/main/karsch.lukas.audit.AuditEntityListener`

entry using `snapshotJson` as the `oldValue` and the current state as the `newValue`.

- **@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` \rightarrow `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`, 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

5.4 Infrastructure

5.5 Performance Tests

6 Results

7 Discussion

7.1 Analysis of results

7.2 Conclusion & Further work

Finally, I'm done!

Glossary

ACID Atomicity, Consistency, Isolation, Durability. 5, 6

Anemic Domain Model The objects describing the domain only hold data, no logic. 3, 4

API API stands for *Application Programming Interface*. It describes the public interface of a module or service, often exposed over a network. 10, 11

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). 5

BASE Basically Available, Soft State, Eventual Consistency. 6

Contract Test A contract test verifies that services implement a shared interface by testing their interactions against an explicitly defined contract. 8

CQRS Command Query Responsibility Segregation. 5–8, 10, 12

CQS Command And Query Separation. 5, 6

CRUD Create Read Update Delete. 5, 8, 10, 11, 20

DAO Data Access Object. 2

DDD Domain Driven Design. 3, 12

DSL Data Specific Language. 13

DTO Data Transfer Object. 6

ES Event Sourcing. 8, 10

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 (*The Apache Groovy programming language - Processing XML* 2026). 13

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 (*The Apache Groovy™ programming language* 2026). 13

HATEOAS Hypermedia as the engine of application state. 1

HTML HyperText Markup Language. 2

HTTP HTTP stands for *Hypertext Transfer Protocol*. It is a protocol used in internet communication and was defined in RFC 2616 (*RFC 2616: HTTP/1.1* 2025). 1, 10, 13–15, 25

JMX Java Management Extensions. 14

JPA Jakarta Persistence API. 11, 17–19

JPQL Java Persistence Query Language. 11

JSON JavaScript Object Notation. 2, 12, 21, 22

ORM Object-relational Mapper. 11

POJO Plain Old Java Object. 12

REST REST stands for *Representational State Transfer*. It is an architectural style for distributed hypermedia systems. 1

REST Assured A library for testing servers. 13, 15

Rich Domain Model Objects incorporate both data and the behavior or rules that govern that data. 3, 4

Testcontainer Testcontainers are a way to declare infrastructure dependencies as code using Docker (*Testcontainers* 2026). 16

URI Uniform Resource Identifier. 1

WWW World Wide Web. 1

XML Extensible Markup Language. 2, 8, 11

References

- ATIS Committee (Mar. 2013). *ATIS Telecom Glossary - audit trail*. en. URL: <https://web.archive.org/web/20130313232104/http://www.atis.org/glossary/definition.aspx?id=5572> (visited on 01/07/2026).
- Axoniq (2025a). *Axon Server - Event Store & Message Delivery System*. en. URL: <https://www.axoniq.io/server> (visited on 01/12/2026).
- (2025b). *Introduction (5.0)*. URL: <https://docs.axoniq.io/axon-framework-reference/5.0/> (visited on 01/12/2026).
- (2025c). *Introduction (v2025.2)*. URL: <https://docs.axoniq.io/axon-server-reference/v2025.2/> (visited on 01/12/2026).
- (2025d). *Messaging Concepts (4.12)*. URL: <https://docs.axoniq.io/axon-framework-reference/4.12/messaging-concepts/> (visited on 01/12/2026).
- Bauer, Christian (2016). *Java persistence with Hibernate*. en. Second edition. Shelter Island, NY: Manning Publications. ISBN: 978-1-61729-045-9.
- Bernstein, Philip A. & Eric Newcomer (2009). *Principles of transaction processing*. en. 2nd edition. The Morgan Kaufmann series in data management systems. Burlington, MA: Morgan Kaufmann Publishers. ISBN: 978-1-55860-623-4.
- Braun, Susanne et al. (Oct. 2021). “Tackling Consistency-related Design Challenges of Distributed Data-Intensive Systems - An Action Research Study”. en. In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. arXiv:2108.03758 [cs], pp. 1–11. DOI: 10.1145/3475716.3475771. URL: <http://arxiv.org/abs/2108.03758> (visited on 12/27/2025).
- Committee on National Security Systems (Apr. 2010). *National Information Assurance Glossary*. en. URL: https://web.archive.org/web/20120227163121/http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf (visited on 01/07/2026).
- Deinum, Marten, Daniel Rubio, & Josh Long (2023). *Spring 6 Recipes: A Problem-Solution Approach to Spring Framework*. en. Berkeley, CA: Apress. ISBN: 978-1-4842-8648-7 978-1-4842-8649-4. DOI: 10.1007/978-1-4842-8649-4. URL: <https://link.springer.com/10.1007/978-1-4842-8649-4> (visited on 01/12/2026).
- Evans, Eric (2004). *Domain-driven design: tackling complexity in the heart of software*. en. Boston: Addison-Wesley. ISBN: 978-0-321-12521-7.
- FasterXML (Oct. 2025). *Jackson Project Home @github*. original-date: 2011-10-19T05:28:40Z. URL: <https://github.com/FasterXML/jackson> (visited on 01/19/2026).

- Fielding, Roy Thomas (2000). “Architectural Styles and the Design of Network-based Software Architectures”. en. PhD thesis. University of California. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf (visited on 01/08/2026).
- Fowler, Martin (Nov. 2003). *Anemic Domain Model*. URL: <https://martinfowler.com/bliki/AnemicDomainModel.html> (visited on 12/27/2025).
- (Apr. 2004). *Audit Log*. URL: <https://martinfowler.com/eaadev/AuditLog.html> (visited on 11/13/2025).
- (Dec. 2005). *Event Sourcing*. URL: <https://martinfowler.com/eaadev/EventSourcing.html> (visited on 11/13/2025).
- (Feb. 2017). *What do you mean by “Event-Driven”?* URL: <https://martinfowler.com/articles/201701-event-driven.html> (visited on 11/13/2025).
- Gray, Jim et al. (1996). “The dangers of replication and a solution”. en. In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. DOI: 10.1145/233269.233330. URL: <https://dl.acm.org/doi/epdf/10.1145/233269.233330> (visited on 12/28/2025).
- Jackson JSON processor* (2026). en-US. concept. URL: <https://docs.oracle.com/en/middleware/goldengate/core/23/ogglc/jackson-json-processor.html> (visited on 01/19/2026).
- Jacobs, Ian & Norman Walsh (Dec. 2004). *Architecture of the World Wide Web, Volume One*. URL: <https://www.w3.org/TR/webarch/> (visited on 12/27/2025).
- Java Programming - The State of Developer Ecosystem in 2023 Infographic* (2026). en. URL: <https://www.jetbrains.com/lp/devecosystem-2023> (visited on 01/12/2026).
- Joint Task Force Interagency Working Group (Sept. 2020). *Security and Privacy Controls for Information Systems and Organizations*. en. Tech. rep. Edition: Revision 5. National Institute of Standards & Technology. DOI: 10.6028/NIST.SP.800-53r5. URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf> (visited on 01/07/2026).
- JUnit User Guide* (2026). URL: <https://docs.junit.org/5.14.2/overview.html> (visited on 01/08/2026).
- Lima, Stanley et al. (June 2021). “Improving observability in Event Sourcing systems”. In: *Journal of Systems and Software* 181, p. 111015. DOI: 10.1016/j.jss.2021.111015.
- Malyi, Roman & Pavlo Serdyuk (Aug. 2024). “Developing a Performance Evaluation Benchmark for Event Sourcing Databases”. en. In: *Visnik Nacìonal’noho unìversitetu "L’vivs’ka polìtehnìka". Serìà Ìnformacìjnì sistemi ta merežì* 15, pp. 159–168. ISSN: 2524065X, 26630001. DOI: 10.

- 23939/sisn2024.15.159. URL: <https://science.lpnu.ua/sisn/all-volumes-and-issues/volume-15-2024/developing-performance-evaluation-benchmark-event> (visited on 11/03/2025).
- Martin, James (1983). *Managing the data-base environment*. en. Englewood Cliffs, N.J.: Prentice-Hall. ISBN: 978-0-13-550582-3.
- Metrics :: Spring Boot* (2026). URL: <https://docs.spring.io/spring-boot/reference/actuator/metrics.html#actuator.metrics.export.prometheus> (visited on 01/19/2026).
- Meyer, Bertrand (2006). *STANDARD EIFFEL*. en.
- Michelson, Brenda (Feb. 2006). *Event-Driven Architecture Overview*. en. Tech. rep. 681. Boston, MA: Patricia Seybold Group, p. 681. DOI: 10.1571/bda2-2-06cc. URL: <http://www.customers.com/articles/event-driven-architecture-overview> (visited on 01/02/2026).
- Micrometer Prometheus :: Micrometer* (2026). URL: <https://docs.micrometer.io/micrometer/reference/implementations/prometheus> (visited on 01/19/2026).
- Overview | Prometheus* (2026). en. URL: <https://prometheus.io/docs/introduction/overview/> (visited on 01/19/2026).
- PostGIS* (2026). en. URL: <https://postgis.net/> (visited on 01/12/2026).
- PostgreSQL Global Development Group (Jan. 2026). *PostgreSQL*. en. URL: <https://www.postgresql.org/> (visited on 01/12/2026).
- Production-ready Features :: Spring Boot* (2026). URL: <https://docs.spring.io/spring-boot/reference/actuator/index.html> (visited on 01/19/2026).
- Prometheus - Monitoring system & time series database* (2026). en. URL: <https://prometheus.io/> (visited on 01/19/2026).
- REST Assured Documentation* (2026). en. URL: <https://github.com/rest-assured/rest-assured/wiki/Usage> (visited on 01/08/2026).
- RFC 2616: HTTP/1.1* (2025). URL: <https://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf> (visited on 12/27/2025).
- Richards, Mark (2015). *Software Architecture Patterns*. en. O'Reilly. ISBN: 978-1-4919-2424-2. URL: <https://theswissbay.ch/pdf/Books/Computer%20science/O'Reilly/software-architecture-patterns.pdf> (visited on 01/08/2026).
- Testcontainers* (2026). en-us. URL: <https://testcontainers.com/> (visited on 01/08/2026).
- The Apache Groovy programming language - Processing XML* (2026). URL: https://groovy-lang.org/processing-xml.html#_gpath (visited on 01/08/2026).
- The Apache Groovy™ programming language* (2026). URL: <https://groovy-lang.org/> (visited on 01/08/2026).

- Tilkov, Stefan (Dec. 2007). *A Brief Introduction to REST*. en. URL: <https://www.infoq.com/articles/rest-introduction/> (visited on 01/08/2026).
- U.S. Securities and Exchange Commission (Aug. 2012). *17 CFR § 242.613 - Consolidated Audit Trail*. en. URL: <https://www.law.cornell.edu/cfr/text/17/242.613> (visited on 01/07/2026).
- Vogels, Werner (Jan. 2009). “Eventually consistent”. en. In: *Communications of the ACM* 52.1, pp. 40–44. DOI: 10.1145/1435417.1435432. URL: <https://dl.acm.org/doi/epdf/10.1145/1435417.1435432> (visited on 12/28/2025).
- Walls, Craig (2016). *Spring Boot in Action*. en. New York: Manning Publications Co. LLC. ISBN: 978-1-61729-254-5.
- Why Spring* (2026). en. URL: <https://spring.io/why-spring> (visited on 01/12/2026).
- Young, Greg (2010). *CQRS Documents by Greg Young*. en. URL: https://cQRS.wordpress.com/wp-content/uploads/2010/11/cQRS_documents.pdf (visited on 12/26/2025).

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>