**Comparative Analysis of MySQL and MongoDB in Developing a High-Concurrency Ticketing System: A Practical Implementation Study**

Blue text on a white background

Description automatically generated

**Gabriel Alberto Avina Solares**

Being a thesis presented for the award of

*Masters Software Solutions Architecture*

Supervisor: Cormac Keogh

Technological University Dublin, Tallaght Campus

School of Enterprise Computing and Digital Transformation

December 2024

**DECLARATION**

I hereby certify that the material, which I now submit for assessment on the programmes of study leading to the award of Master of Science, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Signature of Candidate Date

**ABSTRACT**

**ACKNOWLEDGEMENTS**

**To Silvia, Karla, and Andrea**

**To my friends and supervisor**

**A mis abuelos sláinte mhaith…**

Table of Contents

[**LIST OF TABLES AND FIGURES** 1](#_Toc182676902)

[**ABBREVIATIONS** 2](#_Toc182676903)

[**INTRODUCTION** 3](#_Toc182676904)

[Background and context 3](#_Toc182676905)

[Problem statement 3](#_Toc182676906)

[Research objectives 4](#_Toc182676907)

[Research questions 4](#_Toc182676908)

[Scope and limitations 5](#_Toc182676909)

[Significance of the Study 5](#_Toc182676910)

[1 Review of Relevant Literature 6](#_Toc182676911)

[1.1 Database management systems overview 6](#_Toc182676912)

[1.2 Relational Databases and MySQL 6](#_Toc182676913)

[1.3 NoSQL Databases and MongoDB 6](#_Toc182676914)

[1.3.1 Overview and key features 6](#_Toc182676915)

[1.3.2 Document-oriented databases 9](#_Toc182676916)

[1.4 Transactional Processing and Concurrency 11](#_Toc182676917)

[1.5 Comparative Studies 11](#_Toc182676918)

[1.5.1 Benchmarking Cloud Serving Systems with YCSB (Cooper *et al.*, 2010) 13](#_Toc182676919)

[1.5.2 Databases vs NoSQL Databases (Stonebraker, 2010) 15](#_Toc182676920)

[2 Methodology 15](#_Toc182676921)

[2.1 Research design and approach 16](#_Toc182676922)

[2.2 Description of the Ticketing System Scenario 17](#_Toc182676923)

[2.3 System requirements analysis 17](#_Toc182676924)

[2.3.1 Technical requirements 17](#_Toc182676925)

[2.3.2 Functional requirements 18](#_Toc182676926)

[2.3.3 Non-functional requirements 19](#_Toc182676927)

[2.3.4 MoSCOW Analysis 19](#_Toc182676928)

[2.4 Research validation strategy 20](#_Toc182676929)

[2.5 Testing methodology 20](#_Toc182676930)

[2.6 Data analysis methods 20](#_Toc182676931)

[2.7 Experimental Setup 20](#_Toc182676932)

[3 Implementation 21](#_Toc182676933)

[3.1 MySQL implementation 21](#_Toc182676934)

[3.1.1 Schema Design 22](#_Toc182676935)

[3.1.2 Transaction management 23](#_Toc182676936)

[3.1.3 Concurrency control 24](#_Toc182676937)

[3.2 MongoDB implementation 28](#_Toc182676938)

[3.2.1 System Architecture & Technical Stack 28](#_Toc182676939)

[3.2.2 Document Model Design 29](#_Toc182676940)

[3.2.3 Transaction management 32](#_Toc182676941)

[3.2.4 Concurrency Control 32](#_Toc182676942)

[4 Results and discussion 33](#_Toc182676943)

[4.1 Performance Analysis 33](#_Toc182676944)

[4.1.1 Concurrency Test Results 33](#_Toc182676945)

[4.1.2 Transaction Processing Metrics 33](#_Toc182676946)

[4.2 Schema Analysis 33](#_Toc182676947)

[4.3 Data Structure Management 34](#_Toc182676948)

[4.4 Comparative analysis 34](#_Toc182676949)

[4.4.1 Transaction Management 34](#_Toc182676950)

[4.4.2 Schema Flexibility 34](#_Toc182676951)

[4.4.3 Development Experience 34](#_Toc182676952)

[4.5 Research Questions Assessment 34](#_Toc182676953)

[4.6 Implications 34](#_Toc182676954)

[4.7 Limitations of the research 34](#_Toc182676955)

[5 Conclusion 34](#_Toc182676956)

[5.1 Summary of findings 34](#_Toc182676957)

[5.2 Answers to Research questions 34](#_Toc182676958)

[5.3 Contributions 34](#_Toc182676959)

[5.4 Recommendations and future research 35](#_Toc182676960)

[6 References 35](#_Toc182676961)

[7 Appendices 38](#_Toc182676962)

[7.1 MySQL Schema Implementation 38](#_Toc182676963)

[7.2 Entity-Relationship Diagram 39](#_Toc182676964)

[7.3 Class Diagram 40](#_Toc182676965)

[7.4 Document Structure Diagram for MongoDB 41](#_Toc182676966)

[7.5 Ticket Purchase process diagram 42](#_Toc182676967)

[7.6 Sequence Diagram for Ticket Purchase Transaction 43](#_Toc182676968)

[7.7 Backend application diagram 45](#_Toc182676969)

[7.8 Deployment with technological stack 46](#_Toc182676970)

[7.9 Component Diagram 47](#_Toc182676971)

[7.10 MySQL MoSCOW 48](#_Toc182676972)

[7.11 MongoDB MoSCOW 49](#_Toc182676973)

[7.12 Code samples 50](#_Toc182676974)

[7.13 Test results 50](#_Toc182676975)

[7.14 Configuration files 50](#_Toc182676976)

[7.15 50](#_Toc182676977)

[7.16 Embedding implementation 51](#_Toc182676978)

[7.17 Implementation details 52](#_Toc182676979)

# **LIST OF TABLES AND FIGURES**

# **ABBREVIATIONS**

|  |  |
| --- | --- |
| Business Intelligence | BI |
|  | BERT |
|  | CRISP |
| Online analytical processing | OLAP |
| Extracting, transforming, and loading | ETL |
| Artificial Intelligence | AI |
| Internet of Things | IoT |
| User-generated context (), | UGC |
| Machine Learning | ML |
| Deep Learning | DL |
| Natural Language Processing | NLP |
| Linked Open Data | LOD |
| Structural Topic Modelling | STM |
| JavaScript object notation | JSON |
| binary-encoded serialization of JSON | BSON |
| Extensible Marup Language | XML |
| model-driven architecture | MDA |
| computation-independent model | CIM |
| Platform independent model | PIM |
| platform-specific model | PSM |
| platform independent data metamodel | PIDM |
|  | ORM |
|  | ODM |

# **INTRODUCTION**

## Background and context

*Background of the study refers to the context, circumstances, and history that led to the research problem or topic being studied. It provides the reader with a comprehensive understanding of the subject matter and the significance of the study.*

## Problem statement

The increasing complexity of data management in modern applications presents significant challenges in selecting the appropriate database system. Traditional relational databases like MySQL offer strong transactional integrity and enforce rigid schema designs, making them suitable for applications that require strict data consistency and complex transactions. (Patil *et al.*, 2017),

Conversely, NoSQL databases such as MongoDB have emerged to address the limitations of relational databases by offering flexible schemas, horizontal scalability, and efficient handling of large volumes of unstructured or semi-structured data (Sudiartha *et al.*, 2020). This makes them strong candidates for applications like property listing management systems, which deal with nested data structures, user-generated content and rapid evolving data requirements.

Despite the availability of both SQL and NoSQL solutions, selecting the most appropriate database system for specific application requirements remains a complex task (Capris *et al.*, 2022; Shareef, Sharif and Rashid, 2022; Yedilkhan *et al.*, 2023). Existing comparative studies often provide broad overviews without delving into the practical implications of database selection based on workload characteristics and application contexts (Győrödi *et al.*, 2015). While some research highlights the superior performance of NoSQL databases in handling data loads and dynamic datasets (Chang and Chua, 2019; Sudiartha *et al.*, 2020; Wodyk and Skublewska-Paszkowska, 2020), relational databases continue to be preferred in scenarios requiring strong data integrity and complex transactions.

Developers and organizations need detailed insights to make informed decisions when selecting a database system that balances transactional integrity, development agility, schema design flexibility, and the management of complex data relationships This research aims to address this need by implementing a ticketing system using both MySQL and MongoDB to critically analyse their transactional mechanisms, schema design patterns, and strategies for modelling complex data structures.

By simulating concurrent purchase attempts and managing intricate data relationships within the ticketing system, the study seeks to provide practical insights into the development experiences and challenges associated with each database.

## Research objectives

**Primary objective**

To critically analyse and compare the transactional behaviour, schema design, and management of nested data structures in MySQL and MongoDB through the implementation of a ticketing system.

**Secondary objectives**

1. To examine how MySQL handles transactional integrity and schema rigidity in the ticketing system, especially during simultaneous ticket purchase attempts, and its impact on managing nested data structures.
2. To explore MongoDB’s approach to transactional behaviour and schema flexibility in the same ticketing scenario, analysing how it manages concurrent transactions and the implications of its flexible schema design on modelling intricate data structures.
3. To compare the development experiences and challenges encountered when implementing the ticketing system in both MySQL and MongoDB, focusing on transaction management, agility versus rigidity of schema design, and the strategies for representing of complex structures in each database.

## Research questions

1. How does MySQL ensure transactional integrity in a high-concurrency ticketing system, and what challenges arise from its rigid schema when dealing with complex data relationships?
2. How does MongoDB handle transactions in a concurrent purchase scenario, and how does its flexible schema influence the modelling of intricate data structures?
3. What are the key differences in implementing transactional operations and data modelling between MySQL and MongoDB in the context of the ticketing system?

## Scope and limitations

## Significance of the Study

# Review of Relevant Literature

## Database management systems overview

## Relational Databases and MySQL

It would be nice to have the architecture, data models, strengths, and limitations`. ACID properties and their importance in data integrity

## NoSQL Databases and MongoDB

### Overview and key features

The term NoSQL (Not Only SQL) was first used in 1998 by Carlo Strozzi as the name of his small RDBMS that did not use SQL for data manipulation. Starting in 2009, the term NoSQL is used for the growing number of distributed data management systems that abandoned the support of ACID transactions (Atomicity, Consistency, Isolation, Durability) which is a key principle of relational databases (Lith and Mattsson, 2010).

NoSQL databases have emerged as a pivotal solution for managing the vast and varied data generated in contemporary applications, particularly those involving unstructured and semi structured data. Unlike traditional RDBMS, NoSQL systems offer enhanced scalability, flexibility, and performance, making them well-suited for modern data intensive environments.

These databases are broadly categorized into four primary types on their data models: key-value stores, document-oriented databases, column-family stores, and graph databases. Each one is tailored to specific data management needs, offering unique features that address the limitations of relational databases (Kuznetsov and Poskonin, 2014).

* + 1. Key-Value Stores

Key-value stores are the simplest form of NoSQL databases, where data is stored as a collection of key-value pairs. Each key serves as a unique identifier for its corresponding value, which can be any arbitrary data type (Hecht and Jablonski, 2011; Amghar, Cherdal and Mouline, 2020). This model excels in scenarios requiring high-speed read and write operations, such as caching, session management, and real-time data processing.

The main characteristics of these databases are:

* + - * *Simplicity*: Key-value stores operates on a schema-less model where data is accessed by unique keys, making operations straightforward and efficient but is unsuitable for complex queries, as values are opaque to the system and require logic in the application layer to handle relationships between data (Kuznetsov and Poskonin, 2014).
      * *Performance:* Optimized for speed, particularly in in-memory store like Redis, which offers rapid data access (Amghar, Cherdal and Mouline, 2022).
      * *Scalability:* Easy scalable horizontally by distributing key-value pairs across multiple nodes.

Examples of Key-Value stores databases are Aerospike, Amazon DynamoDB, Memcached, Redis, Berkeley DB, etc.

* + 1. Document-Oriented Databases

According to Kuznetsov and Poskonin (2014) provide richer capabilities than key-value systems. The unit of storage is a document, and the format used is typically JSON, BSON or XML. This model provides indexing and aggregation capabilities, which are advantageous in handling unstructured data.

The main characteristics of these databases are:

* + - * *Schema flexibility:* Documents can have varying structures, accommodating diverse and evolving data models without requiring predefined schemas.
      * *Querying:* Support for complex queries, including indexing and aggregation, facilitates efficient data retrieval and analysis.
      * *Integration with analytical tools:* Compatibility with frameworks like Apache Spark enhances their utility in data-driven applications.

According to Amghar et al.(2022) document-oriented databases are particularly well-suited for applications requiring real-time analytics and flexible data models, such as content management systems and social media platforms.

Some examples of these databases are MongoDB, Couchbase, CouchDB, Realm, Google Cloud Firestore, etc.

* + 1. Column-Family Stores

Inspired by Google’s Bigtable, organize data into columns and rows, grouping related columns into families. This design optimizes storage efficiency and allows for high scalability across distributed systems (Kuznetsov and Poskonin, 2014; Amghar, Cherdal and Mouline, 2020).

The main characteristics of these databases are:

* + - * *Scalability:* Designed for horizontal scalability, handling massive datasets across multiple nodes with ease.
      * *Efficient Storage:* The sparse, multidimensional data model reduces storage inefficiencies by eliminating the need to store null values (ul Haque, Mahmood and Ikram, 2019).
      * *High Throughput:* Suitable for applications requiring high read and write throughput, such as event logging and large-scale analytics.

Examples of Column-family stores are Apache Cassandra, Apache HBase, ScyllaDB, etc.

Despite their strengths, column-family stores demand careful considerations of partitioning strategies to balance performance and flexibility, as highlighted by ul Haque et al. (2019).

* + 1. Graph Databases

Graph databases are designed to manage and query data with complex relationships, using nodes and edges to represent entities and their interconnections. This model is particularly effective for applications where understanding and traversing relationships is crucial. Thakare et al. (2023), Amghar et al., (2022), and Hecht and Jablonski (2011) provided a series of characteristics:

* + - * *Relationship management:* Efficiently handles multidimensional relationships, making them ideal for social networks, recommendation engines, and fraud detection.
      * *Flexible schema:* Supports dynamic and interconnected data structures without requiring a predefined schema.
      * *Advanced querying:* Enables sophisticated queries involving traversals and pattern matching across connected data.

Examples are Neo4j, JanusGraph, MemGraph, etc.

Graph databases, while powerful in managing interconnected data, face challenges in scalability, particularly in distributed environments where graph partitioning is inherently complex.

The exploration of various NoSQL database types reveals their strengths and applicability across different use cases. For the purposes of this research document-oriented databases were chosen, due to their versatility in balancing flexibility, performance and scalability. Their ability to handle complex, semi-structured data and integrate well with analytical tools, makes them ideal for real-time data processing and analytics in the MICE industry where managing diverse data sources is critical for enhancing customer experience and operational efficiency (ul Haque, Mahmood and Ikram, 2019; Amghar, Cherdal and Mouline, 2022).

### Document-oriented databases

Document-oriented databases, a subset of NoSQL databases, are designed to store, retrieve, and manage document-oriented information, typically in formats like JSON, BSON, or XML Unlike traditional relational databases that use tables with rows and columns, document-oriented store data as documents within collections, offering a flexible schema that allows for varying data structures within the same collection (Mok, 2021).

The core of these databases lies in their model and structure. Data is organized into documents, analogous to records, and collections, akin to tables. These documents support nested data structures, including arrays and sub-documents, enabling the representation of complex relationships, within a single entity (Sen and Mukherjee, 2024). The variety of supported data types- strings, numbers, dates, arrays, and sub-documents- enhances their capability to handle diverse data forms.

Schema design in document-oriented databases involves careful consideration of denormalization practices to optimize read performance. The choice between embedding data within documents and referencing other documents depends on factors like data volatility and access patterns. Embedding is often recommended when records are frequently queried together, improving read efficiency, while referencing is preferred for write-heavy workloads or when data is volatile (Imam *et al.*, 2018).

Querying and indexing mechanisms in these databases are powerful and flexible. They support ad-hoc queries, range queries, and aggregations, with indexing on fields within documents improving query performance. Full-text search capabilities further enhance the management of unstructured textual data however, managing schema variations without a rigid schema can complicate data analysis and integration. Gallinucci, Golfarelli, and Rizzi (2018) addressed this challenge by introducing “schema profiling”, a technique that captures and explains schema variations using decision trees, aiding in data quality assessment and business intelligence.

Advantages in managing unstructured data

Scalability and performance are inherent advantages of document-oriented databases. They are designed for horizontal scaling through sharding, distributing data across multiple servers to handle large volumes (Stonebraker, 2010). Load balancing and replication mechanisms enhance performance and availability. However, performance can vary based on the specific database system and workload. For instance, Carvalho, Sá and Bernardino (2023) found that MongoDB outperformed Couchbase and CouchDB in most tests, particularly in read-heavy operations, but may falter in scan-heavy workloads.

Diaz-Ordoñez, Rodríguez Baena and Yun-Casalilla (2023) highlighted the flexibility of these tools due to their dynamic schema, since allows for accommodating evolving data models without downtime or complex migrations, which is essential in industries where data structures frequently change. This flexibility supports rapid development, enabling developers to iterate quickly and adjust data structures as requirements evolve.

Handling diverse data types, particularly unstructured and semi-structured data, is a significant strength. Document databases efficiently store and process data like emails, social media content, multimedia files, and other unstructured formats common in various industries, including MICE, they naturally model complex, hierarchical, and nested data, simplifying data retrieval and manipulation (Sen and Mukherjee, 2024).

Another capability is the performance optimization. Since are tailored for high-performance read and write operations, essential for real-time data access. They support data aggregation and real-time analytics. Mok (2021) demonstrated how schema trees generated from conceptual models can optimize query retrieval times in MongoDB, improving performance in managing large and evolving datasets.

Challenges and limitations

Despite their advantages, these databases present challenges and limitations. According to Imam *et al.* (2018) data integrity concerns arise due to the lack of rigid schemas, potentially leading to inconsistencies. While the flexible schema allows for adaptability, it requires careful management to prevent data anomalies.

Complex transactions are another area of concern investigated by Sen and Mukherjee (2024). Although support for transactions has improved, it may not be as robust as in relational databases for complex, multi-document transactions, also managing write-heavy operations can be challenging, as their schema design focuses on read optimization and may not deeply engage with write-heavy workloads.

Integration with Existing systems

Interoperability is crucial for integrating document databases with existing applications, middleware, and data processing tools. APIs and connectors facilitate this integration, allowing for seamless data flow across systems. Data migration strategies, including ETL processes, are essential for transitioning from RDBMS to document-oriented databases (Li, 2010; Reinero, 2017).

Hybrid approaches are a suitable solution, Seghier and Kazar (2021) presented a polyglot persistence, involving multiple types of databases within the same application to leverage the strength of each, and can be used in combination to optimize performance across different workloads. In a microservices architecture, services can utilize the most appropriate data storage solutions, enhancing scalability and maintainability.

Case studies and Industry examples

Document-oriented databases offer advantages in managing unstructured and semi-structured data, providing flexibility, scalability, and performance optimization. They align well with the needs of industries dealing with complex and evolving data structures, such as the investigated in this proposal.

While challenges exist, particularly in data integrity, complex transactions, learning curve, etc, these can be addressed through careful schema design, validation mechanisms, and ongoing training and tool development.

## Transactional Processing and Concurrency

## Comparative Studies

The evolution of data management systems has led to significant discussions regarding the suitability of traditional RDBMS versus NoSQL databases for handling modern data demands. This section examines the fundamental differences between relational databases and document-oriented NoSQL databases, specifically on their architectures, data models, scalability, performance, and applicability in managing unstructured data.

1. Data models and Schema flexibility

RDBMS structure data into predefined tables with fixed schemas, enforcing strict data integrity and relationships through primary and foreign keys (Stonebraker, 2010; Capris *et al.*, 2022). This rigidity ensures consistency but poses challenges when dealing with unstructured or semi-structured data, as schema alterations can be complex and time consuming (Li, 2010; Digittrix, 2023).

On the other hand, document-oriented NoSQL databases like MongoDB employ flexible, schema-less data models, storing data in JSON-like documents (Thakare *et al.*, 2023). This flexibility allows for the seamless integration of diverse and evolving data types without the need for predefined schemas. Sudiartha et al. (2020) highlighted that this adaptability is crucial for applications dealing with heterogeneous data, such as mobile-based tourist tracking systems.

1. Scalability and performance

Relational databases traditionally scale vertically by improving hardware, which can lead to uprising costs and limitations (Sudiartha *et al.*, 2020; Digittrix, 2023). While, NoSQL are designed for horizontal scalability, distributing data across multiple servers or nodes, thus efficiently handling large volumes of data and high-traffic environments (Amghar, Cherdal and Mouline, 2022; Thakare *et al.*, 2023).

Performance comparisons have consistently shown that NoSQL databases often outperform relational databases in pecific workloads. (Capris *et al.*, 2022) conducted an empirical study comparing MySQL and MongoDB using the Yahoo! Cloud Serving Benchmark (YCSB). The findings indicated that MongoDB significantly outperformed MySQL across various workloads, particularly in write-intensive scenarios. This performance is attributed to MongoDB’s architecture, which avoids the overhead associated with schema validation and foreign key constraints inherent in SQL databases.

1. Handling unstructured data

The schemas of RDBMS make storing unstructured data challenging, often requiring complex transformations or additional layers such as BLOBs (Li, 2010). According to Reinero (2017) Document-oriented databases natively support unstructured data, allowing for direct storage retrieval without extensive preprocessing.

1. Consistency and transactions

RDBMS adhere to ACID properties, ensuring strict data consistency and reliability, which is essential for applications where data integrity is paramount (Stonebraker, 2010; Digittrix, 2023). On the other hand, NoSQL often follow BASE (Basically Available, Soft state, Eventual consistency) principles, prioritizing availability and partition tolerance over immediate consistency (Thakare *et al.*, 2023).

Stonebraker (2010) critiqued NoSQL databases for compromising on transactional integrity to achieve performance gains. HE argued that the performance benefits are not due to the abandonment of SQL but result from reducing overheads like logging and locking in traditional RDBMS. Digittrix (2023) echoes this sentiment, since MongoDB’s lack of mature multi-document ACID transactions may not be suitable for all scenarios, especially where strong consistency is required

1. Integration and migration challenges

Transitioning from RDBMS to NoSQL systems involves significant challenges, including data migration and schema transformation. Li (2010) highlighted the complexities of transforming relational schemas into NoSQL formats like HBase, emphasizing the need for careful planning and potential operational overhead.

(Candel, Sevilla Ruiz and García-Molina, 2022) addressed this issue by proposing a unified metamodel to integrate relational and NoSQL databases, facilitating schema management and promoting polyglot persistence. Reinero (2017) suggested that understanding the fundamental differences in data modelling is crucial for a successful transition, as MongoDB requires a different approach to schema design compared to RDBMS.

The MICE industry deals with a range of unstructured data types, including customer feedback, social media content, and multimedia files. Document-oriented NoSQL databases like MongoDB offer the flexibility and scalability needed to manage this data effectively (Amghar, Cherdal and Mouline, 2022; Digittrix, 2023).

### Benchmarking Cloud Serving Systems with YCSB (Cooper *et al.*, 2010)

Cooper et al. introduced the Yahoo! Cloud Serving Benchmark (YCSB), a framework designed to evaluate the performance of cloud serving systems, particularly NoSQL databases. Addressing the lack of standardized tools for comparing ckoud databases such as Cassandra, HBase, and sharded MySQL.

The findings showed variability between the systems evaluated, depending on the workload characteristics. Cassandra excelled in write-heavy scenarios due to its sequential disk-write optimization, while MySQL and PNUTS performed better in read-heavy workloads. Additionally, all systems demonstrated scalability, but the efficiency in handling increased loads varied across them. Cassandra and PNUTS exhibited better elasticity under stress compared to HBase, which had more erratic performance as the system scaled.

*Comparative analysis*

NoSQL databases such as Cassandra demonstrate superior performance in write-heavy environments, due to their sequential disk-write architecture. SQL databases, on the other hand, tend to focus on maintaining consistency and strong ACID (Atomicity, Consistency, Isolation, Durability) properties, which can increase response time in distributed settings.

Throuhput, which measures how efficiently a database can handle read/write ioeratuinsm was higher in NoSQL systems like Cassandra and HBase in write-heavy workloads. MySQL, while performing well in read-heavy workloads, encountered more difficulty in handling large-scale, distributed environments due to the overhead associated with maintaining ACID compliance. In terms of scalability, the NoSQL systems demonstrated superior horizontal scaling, which is more suitable for cloud applications, whereas SQL databases like Microsoft SQL Server often scale vertically, requiring more powerful individual servers rather than distributing the load across multiple nodes.

Transactional integrity is another critical aspect of comparison. SQL databases, such as Microsoft SQL Server, provide strong transactional integrity through ACID properties. In contrast, NoSQL systems like MongoDB and Cassandra often trade consistency for availability and partition tolerance, following the principles of CAO theorem. This trade-off allows NoSQL databases to achieve higher availability but whit eventual consistency, which can lead to temporary data inconsistencies during operations.

*Strengths and Weaknesses*

MySQL primary strengths are its mature ecosystem, strong transactional integrity, and efficient read operations. Its long-standing presence in the industry means it has robust tools for query optimization, data integrity, and complex queries. Additionally, MySQL’s ability t handle structured data with a well-defined schema makes it ideal for applications that require strong consistency.

However, MySQL struggled with horizontal scalability and write -intensive workloads. The overhead associated with ensuring ACID compliance, particularly in distributed environments, hindered its ability to handle large-scale, high-volume data environments like cloud-based applications.

On the other hand, in the experiment NoSQL systems resulted to be highly scalable, making them better for cloud envionments that required high availability and partition tolerance. They exceled in write-heavy workloads, where MySQL performance was inferior. However, noSQL systems trade off consistency for availability, and their lack of support for complex queries and multi-row transactions could be limited.

Future research should explore how SQL databases like SQL Server perform in more complex transactional settings compared to NoSQL systems.

### Databases vs NoSQL Databases (Stonebraker, 2010)

Stonebraker’s article evaluated the prevailing arguments that advocate for the adoption of NoSQL databases over traditional SQL (RDBMS) systems, particularly focusing on performance and scalability. The primary objective was to determine whether the performance benefits claimed by NoSQL proponents, such as key-value and document-based databases, are substantial enough to justify transitioning from established relational databases.

Stonebraker concluded that the performance advantages of NoSQL databases are often overstated and the result from the elimination of certain overheads inherent in traditional RDBMS, such as logging, locking, and buffer management (Harizopoulos *et al.*, 2008). He argues that these perceived gains do not stem from any inherent architectural superiority of NoSQL systems. Instead, modern SQL databases have incorporated features like automatic sharding and enhanced OLTP (Online Transaction Processing) performance, which address scalability and performance without compromising the ACID (Atomicity, Consistency, Isolation, Durability) properties essential for transactional integrity. Consequently, Stonebraker anticipates the emergence of high-performance, open-source SQL engines that combine the scalability of NoSQL systems with the robust ACID compliance of traditional SQL databases.

# Methodology

## Research design

This study adopted a comparative experimental research approach (Reisner, 1988), to examine how MySQL and MongoDB handle transactional behaviour, schema rigidity, and nested structures within a ticketing system context The experimental design facilitated direct comparison of database performance under controlled conditions through systematic implementation and testing of identical functionality in both systems.

The selection of a comparative experimental methodology was informed in previous studies (Győrödi *et al.*, 2015; Patil *et al.*, 2017; Capris *et al.*, 2022; Stonebraker and Pavlo, 2024), which demonstrated the effectiveness of controlled testing in revealing performance differences between SQL and NoSQL databases. This approach enabled objective measurement of each database’s capabilities in managing concurrent transactions, maintaining data consistency, and handling data relationships.

This research is based on pragmatism, emphasizing practical outcomes and real-world applicability. By implementing actual systems rather than relying solely on theoretical analysis or simulations, the study aims to produce findings that are directly relevant to practitioners facing similar database selection challenges. This aligned with the recommendations of Shareef, Sharif and Rashid (2022), who identified the need for studies that provide actionable insights into database performance in specific application scenarios.

The experimental framework encompassed three key areas of investigation:

1. Transactional behaviour in concurrent ticket booking scenarios
2. Impact of schema design approaches on system implementation
3. Management of nested data structures in booking records

The comparative analysis was structured through:

1. Controlled test scenarios that examined basic booking operations, transaction handling methods, and schema management approaches
2. Systematic comparison of implementation differences, data handling strategies, and transaction management mechanisms
3. Quantitative measurement of performance metrics and qualitative assessment of development experiences

This methodological framework enabled systematic examination of how each database system addressed the core research objectives through controlled experimentation and structured comparison.

## System requirements

### Technical infrastructure

The technical infrastructure for this comparative study was implemented on a development workstation Windows 11 Pro. The system utilized an Intel Core???? Processor with cores, gb ram opeatingat mz, and ssd for stirage operations. Network connectivity was maintained through a gigabit ethernet connection to minimize latency impacts on database operations

The software environment comprised MySQL Community Server 8.0.35 and MongoDB Community Service 7.0.4. Database management was facilitated through MySQL Workbench 8.0.35 and MongoDB Compass 1.40.4 respectively. The development stack included OpenJDK 17.0.9 for core implementation, with Eclipse IDE 2023-09 serving as the primary development environment.

Cnnnection pooling was implemented using HikariCP for MySQL and the MongoDB native connection pool, both configured with:

* Initial pool size: 10 connections
* Maximum pool size: 50 connections
* Connection timeout: 30 seconds
* Idle timeout: 600 seconds

The test environment leveraged Junit 5.10.1 for unit testing and scenario execution. The testing framework was structured to evaluate four specific scenarios:

1. Transaction consistency in MySQL under concurrent operations
2. Schema modification impacts in MySQL during active operations
3. Nested data handling capabilities in MongoDB
4. Concurrent booking conflict resolution in MongoDB

Maven 3.9.5 managed project dependencies and build automation. Version control was maintained through Git 2.42.0, with project artifacts stored in a private repository. Test data generation utilized Mockaroo’s enterprise API for creating realistic booking scenarios and user profiles.

The system architecture implemented domain-driven design principles (reference), separating concerns across:

* Entity definitions
* Data Access Objects (DAO)
* Service layer
* Test simulation framework

### System Architecture

The system architecture implemented a layered approach, adhering to Separation of Concerns principles (SoC) (Software Architect's Handbook by Joseph Ingeno Released August 2018) while facilitating comparative analysis between MySQL and MongoDB implementations. The architecture comprised three primary layers: data access, service, and simulation framework.

#### Data Access Layer

The data access layer established the foundation for database operations through a set of Data Access Objects (DAOs). These objects encapsulated the database specific implementations while maintaining a consistent interface for both MySQL and MongoDB. The layer included specialized DAOs for managing bookings (BookingDAO), tickets (TicketDAO), users (UserDAO), events (EventDAO), and related entities. For MySQL, the implementation utilized JPA annotations and Hibernate persistence provider, while MongoDB implementation employed Morphia for object-document mapping.

The layer implemented the following components:

* Entity mappings (JPA annotations for MySQL, Morphia annotations for MongoDB)
* CRUD operations for all domain objects
* Transaction management (ACID properties for MySQL, session management for MongoDB)
* Connection pool management using HikariCP for MySQL and MongoDB’s native connection pooling

#### Service Layer

The service layer implemented the business logic and transaction orchestration through distinct service classes:

*BookingService* managed the core booking operations:

* User validation
* Ticket availability verification
* Concurrent booking handling
* Transaction coordination
* Status management

*TicketService* handled ticket-specific operations:

* Inventory management
* Category management
* Price calculations
* Availability updates

*UserService* managed user-related operations:

* Authentication logic
* User profile management
* Booking history tracking

#### Simulation Framework

The simulation framework facilitated the comparative analysis through structured test scenarios. This layer comprised:

1. Scenario specific implementations:
   * Transaction consistency testing (MySQL)
   * Schema modification impact assessment (MySQL)
   * Nested data structure handling (MongoDB)
   * Concurrent operation conflict resolution (MongoDB)
2. Metrics collection mechanisms:
   * Response time measurement
   * Transaction success/failure tracking
   * Concurrency conflict detection
   * Resource utilization monitoring
3. Test data management
   * Initial data population
   * Test scenario setup
   * Data consistency verification
   * Results validation

The architecture employed dependency injection patterns (DI) (**references**) to maintain loose coupling between components, enabling isolated testing of database specific characteristics while maintaining consistent application behaviour across both implementations. The separation of concerns facilitated the collection of comparative metrics without compromising the integrity of the core business logic (**references**).

#### Data initialization

The system’s initialization architecture implemented a structured approach to populate test data across both database implementations. This design addressed the challenge of maintaining consistent test data while accommodating the different data models of MySQL and MongoDB.

The initialization framework resolved three key aspects:

1. Managing different referential integrity approaches between MySQL and MongoDB
2. Ensuring consistent test data population across both databases
3. Maintaining data validity for comparative testing

The framework was structured in three main components:

1. *DataInitializer* class served as the central coordinator, managing the initialization sequence and ensuring proper dependency resolution. This approach was necessary to handle the relationships between entities, particularly in maintaining referential integrity in MySQL while allowing for MongoDB’s more flexible document relationships.
2. *Entity* initialization implemented specialized initializers for each domain entity:
   1. Base data initializers (*GenreInitializer, PerformerInitializer, VenueInitializer)*
   2. Event structure initializers *(EventInitializer, TicketCategoryInitializer)*
   3. Transaction data initializers (*TicketInitializer, UserInitializer, BookingInitializer)*

The initialization sequence followed a predetermined order based on entity relationships, ensuring that dependant data was created only after its prerequisites were established.

#### Domain Entities

The system implemented a comprehensive set of domain entities that adopted the core business objects. These entities established the data structure for both database implementations while accommodating their distinct data modelling approaches:

*Event* entity managed performance event data, maintaining relationships with performers, venues, and ticket categories. The implementation supported multiple performers per event and complex ticket category hierarchies.

*Ticket* entity handled individual ticket information, incorporating status management, category association, and booking relationships. The design supported atomic operations (**references**) for concurrent booking scenarios.

*Booking* entity managed transaction data, maintaining relationships between users, events, and tickets. The implementation incorporated status tracking and payment processing requirements.

*User* entity handled customer information and booking history, supporting authentication and profile management requirements.

Supporting entities included:

* *Genre:* Managed musical style classifications
* *Performer:* Handled artist and performer information
* *Venue:* Managed location and capacity data
* *TicketCategory:* Implemented pricing and seating classification

The entity design accommodated both relational and document-based models for a consistent business logic across implementations through flexible relationships mapping strategies (**references**), the appropriate use of normalization in MySQL implementation, strategic denormalization in MongoDB implementation, and consisten identity management across both platforms (**references)**

### Test environment configuration

### Test scenarios

#### Concurrent booking operations

#### Schema flexibility

#### Transaction management

#### Schema modification

### Data collection method

## Research validation strategy

### Experimental validation

Implementation of comparable booking systems in both MySQL and MongoDB

Controlled test environments to ensure fair comparison

Standardized metrics collection across both implementations

### Data validation

Input validation

Verification of test data consistency

Validation of booking requests

Verification of concurrent user simulation parameters

Output validation

Transaction success/failure rates

Data consistency checks

Verification of no overselling

### Process validation

Documentation of all configuration parameters

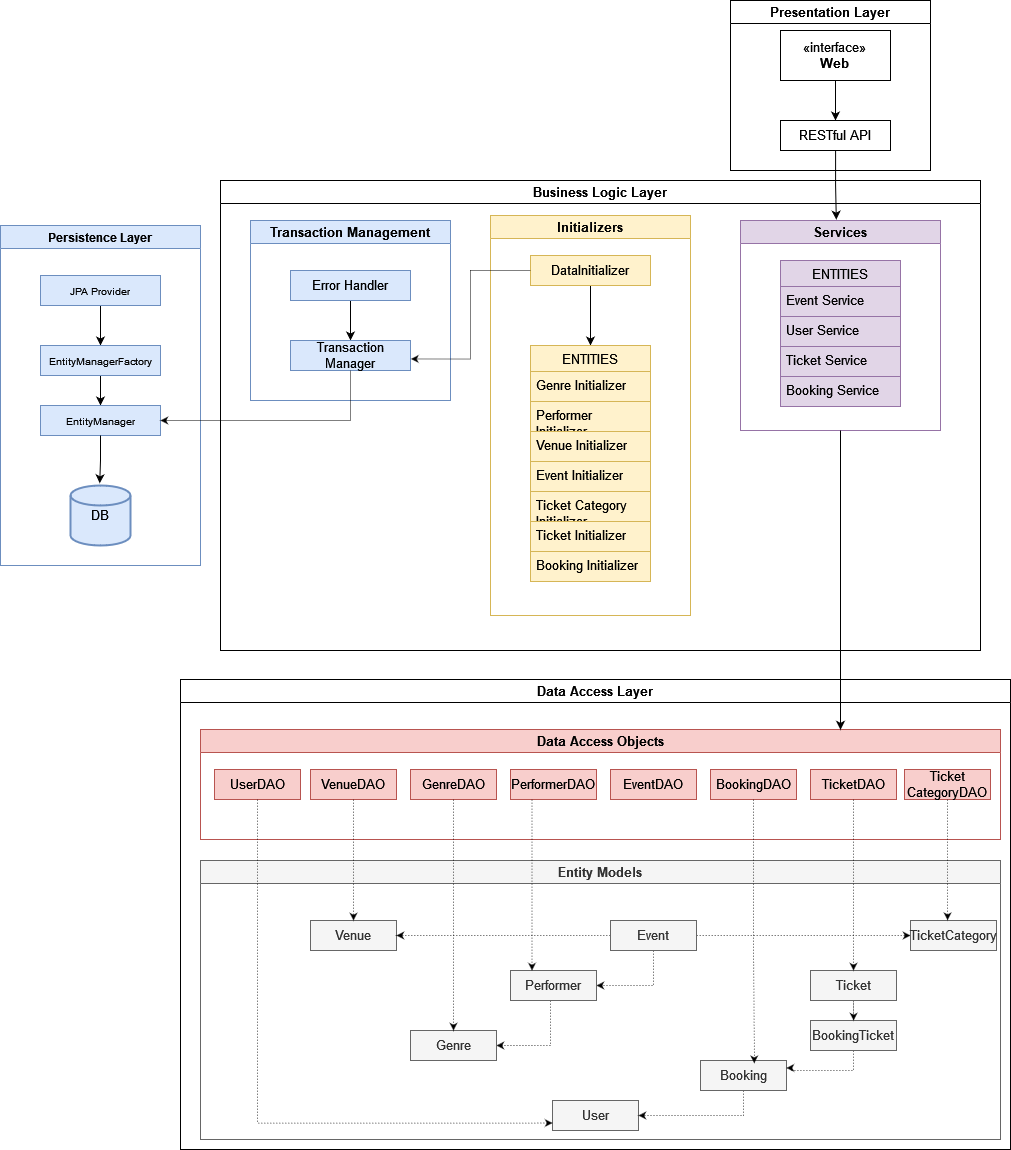
Version control of code

# Implementation

A screenshot of a computer

Description automatically generated

## MySQL implementation



Create a New Maven Project, and configure the dependencies in pom.xml (annex??). JPA dependency, hibernate for JPA implementation with MySQL, MySQL connector, Mongo Java Driver, Junit for Testing, and Log4j for logging. JPA is designed for relational databases, thus integrating with MongoDb requires

Create a Maven project, and selelect the “java-quick-archetype”.

Add a groupid and an artifactid. The version is 0.0.1-SNAPSHOT,

Setting up pom.xml:

Inside the project a pom.xml should be created. Inside the Hibernate and MySQL dependency has been included.

<dependencies>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.4.2.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.22</version>

</dependency>

</dependencies>

Setting up the persistence.xml, connecting it with the local host MySQL. A persistence unit defines the details that are required when you acquire an entity manager.(*Oracle® Fusion Middleware Understanding Oracle TopLink*, 2015)

Database Creation and schema definition

A new MySQL database named TicketSystem was created to host all the application data.

The database schema was defined using SQL “*CREATE TABLE”* statements, afterwards the tables for key entities: *genres, performers, venues, events, ticket\_category, tickets, users, bookings, and booking\_ticket.*

EacH table included the *“AUTO\_INCREMENT”* for unique identification, also constrainst like “*NOT NULL”* and “*UNIQUE”* were applied. Finally the schema (figure 1) as designed in third normal form, to reduce redundancy, and optimize query performance.

### System Architecture & Technical stack

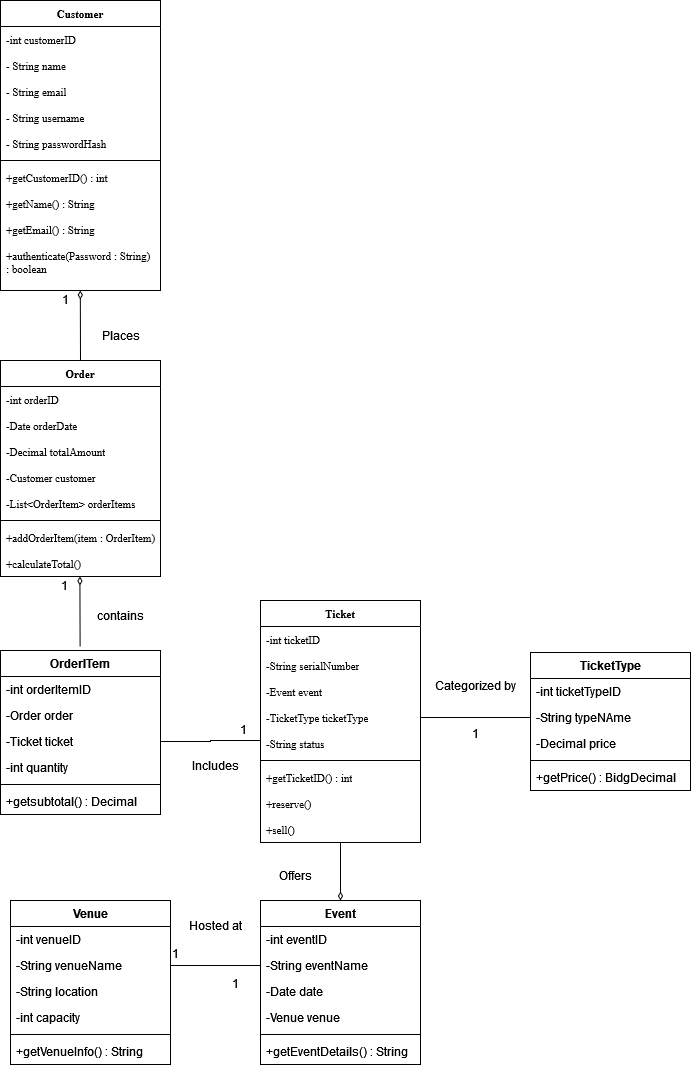
#### MySQL Setup

#### Connection configuration

### Data Model Design

#### Relational Schema Design

#### Transaction management



A screenshot of a computer

Description automatically generated

### Transaction management

 ACID implementation

 Locking mechanisms

 Isolation levels

 Error handling

Enhancing DAOs for Transaction Management and Locking

Before implementing the BookingService, we'll make slight enhancements to your DAOs to support pessimistic locking when reserving tickets. This ensures that when a ticket is being reserved, other transactions cannot modify it until the current transaction is complete. JPA specification defines three pessimistic lock modes that we’re going to discuss:

*findBySerialNumberWithLock* method retrieved a *Ticket* by its serial number and applied a *PESSIMISITC\_WRITE* lock. This for ensured that once a ticket is fetched for reservation, other transactions cannot modify or reserve the same ticket until the current transaction is completed (Kucharz and Valero Sanchez, 2018).

To maintain transactional integrity, all DAO operations within a service used the same *EntityManager*.

Implementing the *BookingService*

The *BookingService* class coordinated the booking process, ensuring that all steps were executed within a single transaction. If any step failed, the entire transaction would be rolled back to maintain data consistency.

**Explanation:**

* **Transactional Control:** Manages transactions manually using EntityTransaction.
* **Pessimistic Locking:** Utilizes the findBySerialNumberWithLock method to apply a PESSIMISTIC\_WRITE lock when reserving tickets.
* **Booking Status:** Directly sets the booking status to CONFIRMED upon successful reservation, bypassing the payment step.
* **Error Handling:** Ensures that any failure during the booking process triggers a transaction rollback, preventing partial data persistence.

 **Transactional Control:**

* The service begins a transaction using EntityManager to ensure that all booking operations are executed atomically.
* If any step fails, the transaction is rolled back to maintain data integrity.

 **Pessimistic Write Locking:**

* The reserveTickets method employs pessimistic locking by invoking ticketDAO.findAvailableTicketsWithLock(eventId, quantity, LockModeType.PESSIMISTIC\_WRITE), which retrieves available tickets with a PESSIMISTIC\_WRITE lock.
* This lock prevents other transactions from accessing the same tickets until the current transaction is completed, thereby avoiding overselling.

 **Data Access Objects (DAOs):**

* **BookingDAO:** Handles the creation and management of Booking entities.
* **TicketDAO:** Manages ticket retrieval and updates, including applying locks during reservation.
* **UserDAO:** Retrieves user information based on user ID.
* **EventDAO:** Fetches event details, including ticket categories and pricing.

 **Booking Process Flow:**

* **User and Event Retrieval:** The service first retrieves the user and event details to validate the booking request.
* **Availability Check:** It then checks if the desired number of tickets is available.
* **Ticket Reservation:** Reserved tickets are locked and marked as RESERVED to prevent concurrent bookings.
* **Price Calculation:** The total price is calculated based on the ticket categories.
* **Booking Creation:** A new Booking record is created and associated with the reserved tickets.
* **Ticket Status Update:** Tickets are updated to SOLD to finalize the booking.

 **Metrics Collection:**

* **successfulBookings:** Tracks the number of successful booking transactions.
* **failedBookings:** Monitors the number of failed booking attempts, aiding in performance and reliability analysis.

 **Error Handling:**

* Comprehensive error handling ensures that any exception during the booking process results in a transaction rollback, maintaining system consistency.
* Specific BookingException instances provide clear feedback on the nature of failures.

**Booking Simulation Class for JPA/MySQL-Based Ticketing System**

The BookingSimulation class is designed to simulate a high-concurrency environment where multiple users attempt to book tickets simultaneously. This simulation is intended to test the effectiveness of the PessimisticBookingService in handling concurrent booking requests without compromising data integrity or allowing ticket overselling.

**Key Objectives:**

1. **Simulate Concurrent Bookings:** Emulate 100 users, each attempting to book 2 tickets for the same event concurrently.
2. **Measure Performance Metrics:** Track the number of successful and failed bookings to evaluate system performance under load.
3. **Ensure Data Consistency:** Verify that the system accurately maintains inventory counts and prevents overselling.

**Class Overview:**

* **Initialization:** Sets up necessary components such as the EntityManager, DAOs, and the PessimisticBookingService.
* **Simulation Execution:** Utilizes a thread pool to manage and execute multiple booking tasks concurrently.
* **Metrics Collection:** Employs thread-safe counters (AtomicInteger) to track successful and failed bookings.
* **Reporting:** Outputs the results of the simulation, including total transactions, success rates, and failure counts.

**Implementation Details:**

1. **Thread Pool Management:** Uses ExecutorService with a fixed thread pool to manage concurrency efficiently.
2. **Task Definition:** Each booking task represents a user attempting to book a specified number of tickets.
3. **Synchronization:** Ensures thread-safe operations when updating shared metrics using AtomicInteger.
4. **Error Handling:** Captures and logs exceptions to prevent simulation crashes and accurately count failed bookings.
5. **Resource Cleanup:** Properly shuts down the thread pool and closes the EntityManager and EntityManagerFactory to release resources.

 Lock management

 Deadlock prevention

 Performance optimization

 Resource management

MySQL Connection Process

Establishing a connection to the MySQL database using JPA involved configuration and troubleshooting to ensure integration. Initially, the setup required defining the database schema with structured tables and relationships to maintain data integrity. Configuring the persistence.xml file was crucial, where specifying accurate connection properties, including the *JDBC URL*, user credentials, and the appropriate *Hibernate* dialect, was essential. A significant challenge encountered was a “*ZoneRulesExceptio*n” caused by an unrecognized time zone ID, which necessitated the addition of a valid “*serverTimezone*” parameter to the JDBC URL to align Java's time zone recognition with the MySQL server settings. Additionally, ensuring that all JPA entity classes were correctly annotated and registered within the persistence unit was important to prevent mapping errors. Another obstacle was rectifying method usage errors, such as substituting *createNamedQuery* with *createQuery* for executing ad-hoc JPQL queries, which resolved exceptions related to undefined named queries. These challenges highlighted the importance of precise configuration and validation of entity mappings.

* Used jakarta.persistence:jakarta.persistence-api
* Used org.hibernate:hibernate-core
* Used mysql:mysql-connector-java

Data population

The implementation utilized the Java Persistence API (JPA) alongside MySQL to initialize data across multiple interconnected tables.

The primary mechanism for data population was the *“DataInitializer”* class, using JPA’s Object-Relational Mapping (ORM) to manage entity relationships and data insertion. Data Access Object (DAO) classes were employed to handle CRUD (Create, Read, Update, Delete) operations for each entity. To prevent data duplication, the initializer utilized methods such as *“findByName()”* within each DAO to verify the existence of records prior to insertion. Additionally, SQL constructs like *“INSERT IGNORE”* were incorporated within native queries to bypass duplicate entries based on predefined unique constraints.

Best Practices to Address Thesis Objectives

Modularity:

Each initializer class handles a specific entity, promoting clean separation of concerns.

Maintainability:

Easier to update or modify data initialization logic for a specific entity without affecting others.

Scalability:

Adding new entities or modifying existing ones requires creating or updating only the relevant initializer class.

Efficiency:

Preloading existing records minimizes database queries, enhancing performance during data initialization.

Transactional Integrity:

Each entity’s data population is encapsulated within its own transaction, ensuring atomicity and reducing the risk of large-scale rollbacks.

CRUD Operations

**Why Not Work Directly with MongoDB Compass or MySQL:**

Instead of working directly with MongoDB Compass or MySQL command-line tools, I focused on backend development to create a controlled environment for testing and comparing the databases. Building the backend with Java and ORM tools allows for:

1. **Automated Data Handling:** Automating data initialization and transaction scenarios ensures consistent and repeatable testing conditions, which is crucial for accurate performance comparisons.
2. **Concurrency Simulation:** Implementing concurrency within the application provides a more realistic simulation of multiple users interacting with the system simultaneously.
3. **Scalability:** A backend application can be easily scaled and modified to test different scenarios without manual intervention, facilitating a comprehensive analysis.
4. **Integration with Both Databases:** Using an Object Relational Mapping framework like Hibernate for MySQL and a similar approach for MongoDB ensures that the comparison focuses on how each database handles similar operations within the same application context.

**Work Completed So Far:**

* **Database Schema Development:**
  + **MySQL:**
    - Designed a normalized relational schema with tables for genres, performers, venues, events, tickets, users, bookings, and their relationships.
    - Enforced data integrity using foreign keys, unique constraints, and NOT NULL constraints.

**Java Implementation:**

* Developed a Java application using Hibernate for MySQL to initialize and populate the database.
* Defined entity classes, Data Access Objects (DAOs), and an initializer to manage data insertion and ensure data integrity.
* Implemented transaction management to maintain consistency in a concurrent environment.

1. Initialization and Data population :
2. Parameter Type mismatch error
3. Enum mapping issue with TicketStatus
4. Infinite Loop during ticket generation

**Duplicate Entry Errors:** Implemented checks to prevent insertion of tickets with duplicate serial numbers, ensuring uniqueness and data integrity.

* Eclipse thrown an *SQLInteritytCOnstaintViolationException* , coming from a duplicate entry in the *serial\_number* field of the tickets table. The error was in the insertion of data, or attempting to insert tickets with serial numbers that already existed in the database.
* *INSERT INTO tickets (event\_id, purchase\_date, rownumber, seat\_number, section, serial\_number, status, ticket\_category\_id)*
* *VALUES (…)*
* *ERROR: Duplicate entry 'ROVI0001' for key 'tickets.serial\_number'*

**Parameter Type Mismatches:** Corrected issues in JPQL queries by ensuring the correct data types are used, which improved query reliability.

* *IllegalArgumentException* occurred because an Event object was passed where an Integer event ID was expected in the findByEventId method of TicketCategoryDAO.

**Enum Mapping Issues:** Created a custom converter to manage differences between database values and Java enums, resolving mapping errors and aligning database records with application logic.

* The system couldn't map database values to Java *enum* constants due to case sensitivity differences (AVAILABLE in Java vs. available in the database), leading to an *IllegalArgumentException*.

**Infinite Loop in Ticket Generation:** Introduced logical limits to prevent infinite loops during ticket creation, enhancing the stability of the data initialization process.

* The seatNumber kept incrementing without bounds. So the solution was to introduce limits for rows and seats within each ticket category and reset the seatNumber for each category.

**Booking-Ticket Associations:** Ensured that associations between bookings and tickets are correctly created and persisted, establishing accurate relationships within the database.

**Transaction Management:** Improved transaction handling to maintain data integrity in a concurrent environment, which is crucial for a high-concurrency system.

try {

em.getTransaction().begin();

// Perform database operations

em.getTransaction().commit();

} catch (Exception e) {

em.getTransaction().rollback();

throw e;

}

## MongoDB implementation

### System Architecture & Technical Stack

#### MongoDB setup

The implementation of MongoDB required a comprehensive setup of both development environment and technical dependencies. The transition from a JPA-based architecture to MongoDB necessitated changes in the project configuration and dependency management.

The development environment setup encompassed several components:

1. MongoDB Community Edition with network service configuration
2. MongoDB Compass for database visualization and management
3. MongoDB Shell (mongosh) for command-line operations

The project’s Maven configuration was updated to incorporate necessary dependencies, primarily:

* MongoDB Driver Sync (version 4.5.1) for database connectivity
* Morphia as the Object-Document Mapping (ODM) (version 2.2.6)

The selection of Morphia as the ODM solution was based on its compatibility with MongoDB’s document model, replacing the traditional JPA implementation that was not inherently designed for NoSQL databases. This decision facilitated more natural mapping between Java objects and MongoDB documents.

#### Connection configuration

The database initialization and connection management were implemented through a *DataInitializer* class, which served as the central component for establishing and managing MongoDB connections. This class utilized Morphia ODM and implemented an approach to database connectivity and data management.

Key implementation aspects included:

* MongoDB connection was established using *MongoClients.create()* with a local connection string (“mongodb://localhost:27017”).
* A Morphia Datastore was created and configures to manage the “ticketsystem” database through *datastore.ensureIndexes().*
* The implementation included automatic index management through *datastore.ensureIndexes()* (Kumar, 2019).
* The *MongoClient* instance was managed to control the connection lifecycle
* Resource cleanup was handled through a dedicated *close()* method
* Connection validation was implemented through data verification routines

public DataInitializer() {

mongoClient = MongoClients.create("mongodb://localhost:27017");

datastore = Morphia.createDatastore(mongoClient, "ticketsystem");

datastore.getMapper().mapPackage("dev.morphia.example");

datastore.ensureIndexes();

}

### Document Model Design

The database employed a combination of embedding and referencing strategies to maintain data consistency across collections:

1. Genres
   1. Stores information about genres linked to performers.
   2. Schema enforces a unique *name* field with indexing to ensure fast query performance and prevent duplicates.
2. Performers
   1. Contain details, with each performer referencing *genre\_id* from the genres collection.
   2. Unique indexes on *name* and *genre\_id* for data integrity and query efficiency.
3. Venues
   1. Hold venue details, with filds like *name, address, type,* and *capacity.*
   2. The schema enforces uniqueness on venue names and requires non-negative capacity values.
4. Events
   1. Encapsulates event details, including embedded ticket categories with attributes like *description, price, start\_date,* and *area.*
   2. The schema also used compound indexes to prevent duplicate events and optimize lookups by performer or venue.
   3. Embedding ticket simplifies data retrieval, as categories are tightly coupled with their events.
5. Users
   1. Represented system users with unique *username* and *email* fields.
   2. Validation for emails followed a valid format, while indexing improved performance and avoided duplicated entries.
6. Booking
   1. Records user ticket purchases, including fields like *user\_id, delivery\_email, status,* and *tickets.*
   2. The tickets array references the *tickets* collection, reducing redundancy and enabling association between bookings and tickets.
   3. Indexes on *user\_id* and *status* supported lookups by user and booking state.
7. Tickets
   1. Stores individual ticket details, with fields like *serial\_number, event\_id, section,* and *status.*
   2. Unique indexes ensure each ticket has a distinct serial number, while compound indexes on *event\_id* and *status* enabled quick retrieval of available tickets.
   3. The ticket category is denormalized to simplify queries.

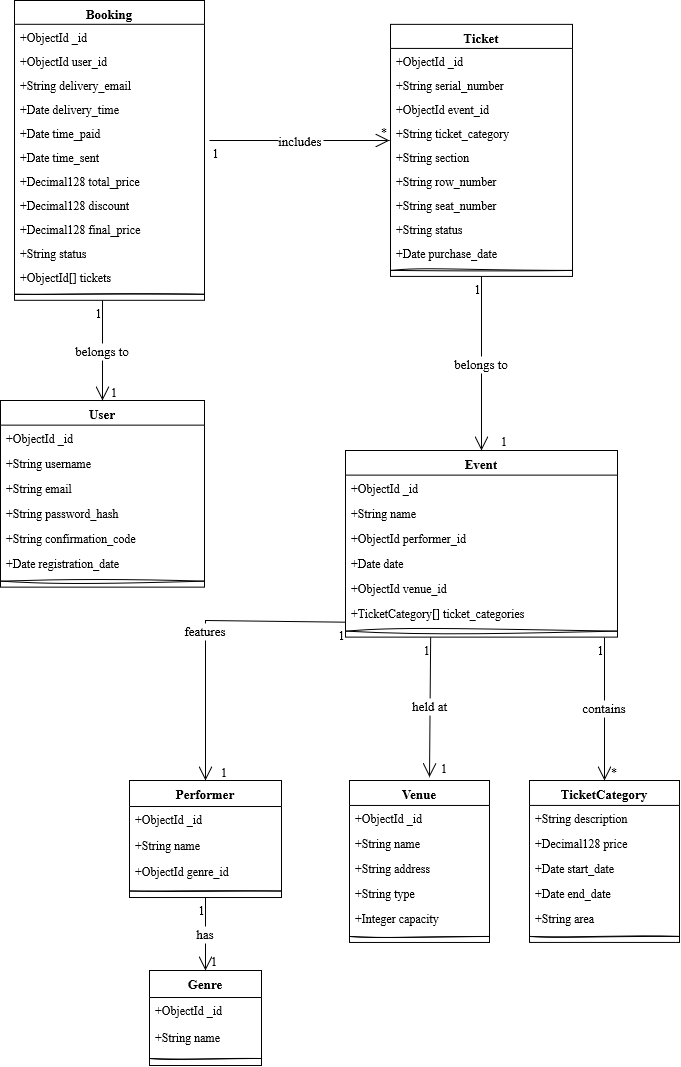


Figure 1. MongoDB System Class Diagram

#### Flexibility vs Rigidity Impact

MongoDB employed a schema-less architecture (Figure 2), allowing each document within a collection to possess a distinct structure. This flexibility facilitated the accommodation of evolving application requirements without necessitating extensive schema migrations. In the context of the ticketing system, this adaptability enabled the embedding of *TicketCategory* documents within *Event* documents, thereby optimizing read performance by minimizing the need for join operations.

A diagram of a function

Description automatically generated

Figure 2. DB Schema Flexible using MongoDB

However, the flexible schema also introduced challenges related to data consistency and redundancy. Without enforced schema constraints, maintaining consistent data structures across documents required meticulous application-level validations. For example, ensuring that all *User* documents contained necessary fields such as *passwordHash* and *confirmationCode* necessitated validation logic with the application layer.

Example:

@Entity("users")

public class User {

@Id

private ObjectId id;

*//unique username for login*

@Property("user\_name") //maps to username

@Indexed(options = @dev.morphia.annotations.IndexOptions(unique = true, name = "userName\_idx"))

private String userName;

*//unique email address*

@Property("email")

@Indexed(options = @dev.morphia.annotations.IndexOptions(unique = true, name = "email\_idx"))

private String email;

@Property("password\_hash")

private String passwordHash;

@Property("confirmation\_code")

private String confirmationCode;

@Property("registration\_date")

private Date registrationDate;

Additionally, denormalization, while beneficial for read performance, increased the potential for data redundancy. Embedding related data could lead to duplicated information if not managed appropriately, thereby increasing storage requirements and complicating update operations. For example, if a *TicketCategory* description needed to be updated, the change would have to be propagated to every embedded instance within *Event* documents to maintain consistency.

@Entity("ticketCategories")

public class TicketCategory {

//@Id

//private Object id;

@Property("description")

private String description;

@Property("price")

private BigDecimal price;

@Property("start\_date")

private Date startDate;

@Property("end\_date")

private Date endDate;

@Property("area")

private String area; // Seating area description

Challenges arose during implementation, particularly in embedding ticket categories within event documents and preventing data duplication. These were resolved by using Morphia annotations, such as replacing the deprecated @*Embedded* with *@Reference* without an *@Id* field for embedded documents (Chooly, 2019). The collection for ticket categories was changed from a *List* to a *Set,* and equality checks were implemented to ensure uniqueness. These adjustments not only resolved immediate issues but also enhanced the system’s ability to handle dynamic data structures without sacrificing consistency.

@Entity("events")

@Indexes({

@Index(fields = {@Field("event\_name"), @Field("date")}, options = @IndexOptions(unique = true, name = "name\_date\_idx")),

@Index(fields = @Field("performer"), options = @IndexOptions(name = "performer\_idx")),

@Index(fields = @Field("venue"), options = @IndexOptions(name = "venue\_idx"))

})

public class Event {

@Id

private ObjectId id; //mongos unique identifier

@Property("event\_name")

private String name;

@Reference(lazy = true)//References the performer document, loaded lazily

private Set<Performer> performers;

@Property("date")

private Date date; //date of the event

@Reference(lazy = true)

private Venue venue; //References the venue document

//("ticketCategories") //embeds a list of TicketCategory documents within the Event document

@Reference("ticketCategories")

private Set<TicketCategory> ticketCategories; //list of ticket categories for the event

The schema proposed offered significant flexibility, particularly by enabling embedding of related documents like *TicketCategory* within *Event* documents to optimize performance. However, this flexibility came with challenges, including the need for an strict application-level validations to ensure consistency and the risk of redundancy inherent in denormalized designs.

#### Data Structure Management

Document-oriented model allows for the natural representation of complex data structures within individual documents. To prevent duplicate entries and ensure uniqueness of critical fields, unique indexes were implemented across various collections. This strategy enforced data integrity, at the database level eliminating the possibility of duplicate records that could compromise system reliability.

Example:

*@Entity*("bookings") // Maps to the 'bookings' collection

*@Indexes*({

*@Index*(fields = *@Field*("user\_id"), options = *@IndexOptions*(name = "user\_id\_idx")),

*@Index*(fields = *@Field*("status"), options = *@IndexOptions*(name = "status\_idx

})

public class Booking {

*@Id*

private ObjectId id;

*@Property*("user\_id")

private ObjectId userId;

// \*\*Add event\_id to Booking\*\*

*@Property*("event\_id")

private ObjectId eventId;

*@Property*("delivery\_email")

private String deliveryEmail;

*@Property*("delivery\_time")

private Date deliveryTime;

*@Property*("time\_paid")

private Date timePaid;

// Status of the booking ('confirmed', 'in-progress', 'canceled')

*@Property*("status")

private String status;

// List of ticket ObjectIds associated with the booking

*@Property*("tickets")

private List<ObjectId> tickets;

The *Booking* entity referenced both *User* and *Ticket* documents, maintaining normalization by avoiding the embedding of extensive ticket arrays within a booking. *@Indexed annotations* ensured that fields maintained unique values. Attempting to insert duplicate values in these fields would result in a database error, thereby preserving data uniqueness.

#### Relationship Handling

One-to-Many relationships implemented through embedding or referencing dependent entities. For example, a *User* could have multiple *Booking* documents, established via a one-to-many relationships where each booking referenced its associated user.

Many-to-many relationships managed using referencing strategies. The relationship between *Performer* and *Genre* entities was handled by referencing genres within performer documents or vice versa, depending on access patterns.

Implementation techniques:

* Morphia annotations:
  + *@Reference:* Facilitated referencing documents without embedding, maintaining loose coupling between entities.
* Data integrity maintenance:
  + Application-level validation ensured referential integrity through application logic due to the lack of enforced foreign key constraints in MongoDB.
  + Indexing creation on referenced fields to optimize join-like operations.

Example:

@Entity("events")

public class Event {

@Id

private ObjectId id;

private String name;

private Date date;

@Reference

private Venue venue; // Referenced for scalability

@Reference

private List<TicketCategory> ticketCategories; //

// Getters and Setters

}

The *Event* entity referenced *TicketCategory* and *Venue* using @*Reference* annotation, enabling the retrieval of related data without embedding entire performer or venue documents within each event.

### Transaction management

The implementation of transaction management represented a critical component of the system, particularly in handling concurrent booking operations while maintaining data consistency. While traditional NoSQL systems often prioritize eventual consistency following the CAP theorem (Brewer, 2012), MongoDB’s introduction of multi-document ACID transactions starting from version 4.0 (O’Grady, 2020) marked a significant shift toward stronger consistency guarantees. This implementation bridges the gap between NoSQL flexibility and traditional RDBMS reliability, following the theoretical framework established by Pritchett, (2008) regarding eventually consistent systems. atomic operations across multiple documents and collections. Although historically NoSQL databases sacrificed strong consistency for availability and partition tolerance, following BASE (Basically Available, Soft state, Eventually consistent) principles

The system developed for this study adopted a hybrid approach that balances ACID compliance with distributed system performance. This aligns with Stonebraker (2010) observations regarding the necessity of maintaining transactional integrity in specific domains, and its implementation was made through a structured booking service that coordinated multiple database operations within atomic units. The transaction management system implemented what Gray and Lamport (2006) term a “distributed transaction commit protocol”, modified for document-oriented databases:

A diagram of a process

Description automatically generated

Figure 3. Transaction Flow

#### Core Transaction component

The core transaction management was implemented through the *BookingService* class. The session-based transaction management with operations across multiple collections:

// *Primary transaction coordinator*

public class BookingService {

private final BookingDAO bookingDAO;

private final TicketDAO ticketDAO;

private final UserDAO userDAO;

private final EventDAO eventDAO;

private final Datastore datastore;

// Metrics tracking

private AtomicInteger successfulBookings = new AtomicInteger(0);

private AtomicInteger failedBookings = new AtomicInteger(0);

}

This implementation followed the Repository Pattern (Fowler, 2002), separating transaction logic from data access concerns, enabling a clean transaction, where the *BookingService* class acted as the primary transaction coordinator, managing multiple Data Access Objects (DAOs) and maintained transaction metrics.

In general terms the architecture exhibited several characteristics:

1. Resource encapsulation
   1. DAO references were immutable and separated, promoting modularity.
   2. Transaction coordinator was centralized within the *BookingService*
2. State management
   1. Atomic operations were used for counting successful and failed bookings (*AtomicInteger).*

#### Transaction protocol

The protocol implemented a multi-phase commit pattern, aligning with the “ACID transaction protocol” (Faraj, 2022), involving distinct phases of resource verification, allocation, state update, and commit, each delineated to ensure atomicity and consistency (Appendix 16).

The lifecycle management implemented a session-based approach to transaction processing. During session initialization, the system first acquired the necessary resources required for processing transactions. This phase established transaction boundaries to encapsulate all operations within a single transactional context. The implementation was supported by Java’s try-with-resources construct which ensured atomic cleanup of resources.

The state management component utilized MongoDB’s *ClientSession* to maintain transaction integrity. Also, the employment of atomic counters (*successfulBookings and failedBookings)* to track transaction outcomes, providing metrics for monitoring booking attempts (Salander, 2022). All operations were confined within their respective session scopes, which maintained isolation and consistency of the transaction data.

The transaction workflow was structured through distinct phase separation. The first phase handed resource verification, where the system validated the event’s existence and checked available ticket quantities.

The second phase manged resource allocation, where the system attempted to book the requested number of available tickets. The third phase handled state updates by creating a new booking record in the system. The final commit phase concluded the transaction by committing all changes to the database.

public boolean bookTickets(ObjectId userId, ObjectId eventId, int quantity) {

try (ClientSession session = datastore.startSession()) {

session.startTransaction();

try {

***// Phase 1: Resource Verification***

Event event = eventDAO.findById(eventId);

long availableTickets = ticketDAO.countAvailableTickets(eventId);

***// Phase 2: Resource Allocation***

List<Ticket> bookedTickets = ticketDAO.bookAvailableTickets(

session, eventId, quantity);

**// Phase 3: State Update**

Booking booking = new Booking(/\* ... \*/);

bookingDAO.create(booking);

***// Phase 4: Commit***

session.commitTransaction();

successfulBookings.incrementAndGet();

return true;

} catch (Exception e) {

***// Rollback Phase***

session.abortTransaction();

failedBookings.incrementAndGet();

return false;

}

}

}

Error handling was implemented through a try-catch mechanism that defined error boundaries, utilizing transaction abort functionality to ensure automatic rollback of all changes in case of failures (Bernstein and Newcomer, 2009). This mechanism guaranteed that the system-maintained consistency by reverting to its previous state when errors occurred, while also incrementing the *failedBookings* counter for monitoring purposes.

#### Core Concurrency component

The *BookingSimulation* class (Appendix 17) embodied the work done by Seppälä (2024) facilitating management of concurrent booking operations through a fixed thread pool and task queuing mechanisms.

The class took in several key dependencies including instances of *BookingDAO, UserDAO, EventDAO, TicketDAO,* and *Datastore,* object from the Morphia library. These dependencies allowed the class to interact with the underlying data storage and retrieve the necessary information to carry out the simulation.

The *runSimulation* method was the main entry point for executing the booking simulation. It accepted three parameters: the ID of the event for which bookings should be simulated, the number of users to simulate booking attempts for, and the maximum number of tickets each user could attempt to book.

1. Creating of a fixed-size thread pool using the *Executors.newFixedThreadPool* method. The size of the thread pool could be adjusted based in the expected concurrency level and system resources.
2. Retrieval of all registered users from the *UserDAO* to serve as the simulated booking agents
3. If the number of users is insufficient for the specified simulation size, a warning is printed, and the simulation is terminated.
4. For each simulated user, a *callable* task is created that invoked the *bookTickets* method of the *BookingService* class. The number of tickets to be booked for each user is randomly determined within the specified maximum.
5. All the *Callable* tasks are submitted to the thread pool using the *invokeAll* method, and the system waits for all tasks to complete.
6. Once the simulation is finished, the results are printed, including the total number of booking attempts, the number of successful bookings, and the number of failed bookings.
7. Finally, the system verifies that the total number of booked tickets does not exceed the total number of available tickets for the event, preventing any potential overselling scenarios.

This experiment aimed to demonstrate the ability to handle a number of concurrent booking requests and provided insights into the effectiveness of the underlying concurrency control mechanisms.

#### Resource manager implementation

The *TicketDAO* class functioned as the primary resource manager, orchestrating the booking of available tickets. A point for highlighting is that this approach employed a resource locking strategy to maintain data integrity as suggested King (2024).

Each ticket’s state was updated individually, adhering to a consistent status update where transitions were uniformly applied across operations, maintaining consistency.

In relation to King's (2024) approach, a resource locking strategy was implemented with an optimistic concurrency control strategy to handle simultaneous transactions, in which the status-based availability was determined based on the ticket status, preventing overbooking.

.filter(Filters.and(

Filters.eq("event\_id", eventId),

Filters.eq("status", "available")

))

Transactions were encapsulated within *ClientSession* blocks to ensure atomicity during operations, such as ticket bookings. The implementation of the ticket booking functionality in the system incorporated an atomic approach to ensure reliable and consistent operations during high-concurrency scenarios. The method *bookAvailableTickets* encapsulated the logic for finding and updating ticket statuses within the system, and it primary task was to retrieve tickets matching specific criteria (those linked to a given event ID and marked as “available”. This retrieval was performed using MongoDB filters within a *datastore.find()* operation, ensuring that only eligible tickets were selected (Chooly, 2024). The method iteratively processed the required quantity of tickets, updating their status to “booked” and recording the purchase date before saving the changes to the database. By using *datastore.save()* for each ticket, the system ensured that updates were atomic, reducing the risk where multiple concurrent transactions might modify the same tickets.

This method successfully handled the requirements of atomicity and concurrency control, ensuring reliable ticket booking in real-time.

# Results and discussion

## Performance Analysis

### Concurrency Test Results

### Transaction Processing Metrics

## Schema Analysis

|  |  |  |
| --- | --- | --- |
| **Aspect** | **MySQL (genres Table)** | **MongoDB (genres Collection)** |
| **Schema Structure** | Relational table with fixed columns and data types. | Document-based with JSON Schema validation. |
| **Primary Key** | genre\_id (INT, AUTO\_INCREMENT). | \_id (ObjectId, automatically generated). |
| **Unique Constraints** | genre\_name is unique via SQL constraint. | name is unique via MongoDB unique index. |
| **Data Integrity** | Enforced through foreign keys and constraints. | Enforced via JSON Schema and application-level validations. |
| **Flexibility** | Rigid schema; altering structure requires migrations. | Flexible schema; easy to add new fields without migration. |
| **Indexing** | Indexed through primary and unique keys. | Indexed using MongoDB's createIndex with uniqueness. |
| **Relationships** | Managed through foreign keys in related tables. | Managed via referencing (genre\_id as ObjectId). |
| **Query Complexity** | Simple queries for single table; joins for relations. | Single collection queries; aggregation needed for relations. |

## Data Structure Management

## Comparative analysis

### Transaction Management

### Schema Flexibility

### Development Experience

## Research Questions Assessment

## Implications

## Limitations of the research

# Conclusion

## Summary of findings

## Answers to Research questions

## Contributions

## Recommendations and future research

# References

Amghar, S., Cherdal, S. and Mouline, S. (2020) ‘Data Integration and NoSQL Systems: A State of the Art’, in *Proceedings of the 4th International Conference on Big Data and Internet of Things*. *International Conference on Big Data and Internet of Things*, New York, NY, USA: Association for Computing Machinery (BDIoT ’19), pp. 1–6. Available at: https://doi.org/10.1145/3372938.3372954.

Amghar, S., Cherdal, S. and Mouline, S. (2022) ‘Storing, preprocessing and analyzing tweets: finding the suitable noSQL system’, *International Journal of Computers and Applications*, 44(6), pp. 586–595. Available at: https://doi.org/10.1080/1206212X.2020.1846946.

Babic, T. (2019) *A Data Model for Online Concert Ticket Sales*, *Vertabelo Data Modeler*. Available at: https://vertabelo.com/blog/a-data-model-for-online-concert-ticket-sales/ (Accessed: 19 October 2024).

Bernstein, P.A. and Newcomer, E. (2009) ‘Chapter 1 - Introduction’, in P.A. Bernstein and E. Newcomer (eds) *Principles of Transaction Processing (Second Edition)*. San Francisco: Morgan Kaufmann (The Morgan Kaufmann Series in Data Management Systems), pp. 1–29. Available at: https://doi.org/10.1016/B978-1-55860-623-4.00001-9.

Brewer, E. (2012) ‘CAP twelve years later: How the “rules” have changed’, *Computer*, 45(2), pp. 23–29. Available at: https://doi.org/10.1109/MC.2012.37.

Candel, C.J.F., Sevilla Ruiz, D. and García-Molina, J.J. (2022) ‘A unified metamodel for NoSQL and relational databases’, *Information Systems*, 104, p. 101898. Available at: https://doi.org/10.1016/j.is.2021.101898.

Capris, T. *et al.* (2022) ‘Comparison of SQL and NoSQL databases with different workloads: MongoDB vs MySQL evaluation’, in *2022 International Conference on Data Analytics for Business and Industry (ICDABI)*. *2022 International Conference on Data Analytics for Business and Industry (ICDABI)*, IEEE, pp. 214–218. Available at: https://doi.org/10.1109/ICDABI56818.2022.10041513.

Carvalho, I., Sá, F. and Bernardino, J. (2023) ‘Performance Evaluation of NoSQL Document Databases: Couchbase, CouchDB, and MongoDB’, *Algorithms*, 16(2), p. 78. Available at: https://doi.org/10.3390/a16020078.

Chang, M.-L.E. and Chua, H.N. (2019) ‘SQL and NoSQL Database Comparison’, in K. Arai, S. Kapoor, and R. Bhatia (eds) *Advances in Information and Communication Networks*. *FICC 2018*, Cham: Springer International Publishing, pp. 294–310. Available at: https://doi.org/10.1007/978-3-030-03402-3\_20.

Chooly, E. (2019) *Version 1.5 Migration Guide · Issue #1337 · MorphiaOrg/morphia*, *GitHub*. Available at: https://github.com/MorphiaOrg/morphia/issues/1337 (Accessed: 16 November 2024).

Chooly, E. (2024) ‘MorphiaOrg/morphia’. MorphiaOrg. Available at: https://github.com/MorphiaOrg/morphia (Accessed: 16 November 2024).

Cooper, B.F. *et al.* (2010) ‘Benchmarking cloud serving systems with YCSB’, in *Proceedings of the 1st ACM symposium on Cloud computing*. *1st ACM symposium on Cloud computing (SoCC ’10)*, New York, NY, USA: Association for Computing Machinery (SoCC ’10), pp. 143–154. Available at: https://doi.org/10.1145/1807128.1807152.

Diaz-Ordoñez, M., Rodríguez Baena, D.S. and Yun-Casalilla, B. (2023) ‘A new approach for the construction of historical databases—NoSQL Document-oriented databases: the example of AtlantoCracies’, *Digital Scholarship in the Humanities*, 38(3), pp. 1014–1032. Available at: https://doi.org/10.1093/llc/fqad033.

Digittrix, N. (2023) ‘The Battle of Databases: MySQL vs. MongoDB’, *Medium*, 18 September. Available at: https://medium.com/@nikita.digittrix/the-battle-of-databases-mysql-vs-mongodb-ca57164b8386 (Accessed: 22 September 2024).

Faraj, H.A. (2022) ‘Moving RDBMS to NoSQL Paradigms’, in *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*. *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA: IEEE, pp. 684–689. Available at: https://doi.org/10.1109/CSCI58124.2022.00125.

Fowler, M. (2002) *Patterns of Enterprise Application Architecture*. Wesley. Available at: https://dl.ebooksworld.ir/motoman/Patterns%20of%20Enterprise%20Application%20Architecture.pdf.

Gallinucci, E., Golfarelli, M. and Rizzi, S. (2018) ‘Schema profiling of document-oriented databases’, *Information Systems*, 75, pp. 13–25. Available at: https://doi.org/10.1016/j.is.2018.02.007.

Gray, J. and Lamport, L. (2006) ‘Consensus on transaction commit’, *ACM Trans. Database Syst.*, 31(1), pp. 133–160. Available at: https://doi.org/10.1145/1132863.1132867.

Győrödi, C. *et al.* (2015) ‘A comparative study: MongoDB vs. MySQL’, in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*. *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, pp. 1–6. Available at: https://doi.org/10.1109/EMES.2015.7158433.

ul Haque, A., Mahmood, T. and Ikram, N. (2019) ‘Performance Comparison of State of Art NoSql Technologies Using Apache Spark’, in K. Arai, S. Kapoor, and R. Bhatia (eds) *Intelligent Systems and Applications*. *Intelligent Systems and Applications. IntelliSys 2018*, Cham: Springer International Publishing, pp. 563–576. Available at: https://doi.org/10.1007/978-3-030-01057-7\_44.

Harizopoulos, S. *et al.* (2008) ‘OLTP through the looking glass, and what we found there’, in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: Association for Computing Machinery (SIGMOD ’08), pp. 981–992. Available at: https://doi.org/10.1145/1376616.1376713.

Hecht, R. and Jablonski, S. (2011) ‘NoSQL evaluation: A use case oriented survey’, in *2011 International Conference on Cloud and Service Computing*. *2011 International Conference on Cloud and Service Computing*, Hong Kong, China: IEEE, pp. 336–341. Available at: https://doi.org/10.1109/CSC.2011.6138544.

Imam, A.A. *et al.* (2018) ‘Data Modeling Guidelines for NoSQL Document-Store Databases’, *International Journal of Advanced Computer Science and Applications (ijacsa)*, 9(10). Available at: https://doi.org/10.14569/IJACSA.2018.091066.

King, E. (2024) *Hello Interview | System Design in a Hurry*, *Hello Interview*. Available at: https://www.hellointerview.com/learn/system-design/answer-keys/ticketmaster (Accessed: 23 October 2024).

Kucharz, P. and Valero Sanchez, J. (2018) *Pessimistic Locking in JPA | Baeldung*. Available at: https://www.baeldung.com/jpa-pessimistic-locking (Accessed: 17 November 2024).

Kumar, C. (2019) *Introduction to Morphia - Java ODM for MongoDB | Baeldung*, *Baeldung*. Available at: https://www.baeldung.com/mongodb-morphia (Accessed: 16 November 2024).

Kuznetsov, S.D. and Poskonin, A.V. (2014) ‘NoSQL data management systems’, *Programming and Computer Software*, 40(6), pp. 323–332. Available at: https://doi.org/10.1134/S0361768814060152.

Li, C. (2010) ‘Transforming relational database into HBase: A case study’, in *2010 IEEE International Conference on Software Engineering and Service Sciences*. *2010 IEEE International Conference on Software Engineering and Service Sciences*, pp. 683–687. Available at: https://doi.org/10.1109/ICSESS.2010.5552465.

Lith, A. and Mattsson, J. (2010) *Investigating storage solutions for large data - A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data*. CHALMERS UNIVERSITY OF TECHNOLOGY. Available at: https://www.semanticscholar.org/paper/Investigating-storage-solutions-for-large-data-A-of-Lith-Mattsson/b8f6b9d79c75e66c3b2f5034fe8172fd24cc0d13 (Accessed: 28 August 2024).

Mok, W.Y. (2021) ‘A Logical Database Design Methodology for MongoDB NoSQL Databases’, in *2021 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*. *2021 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, Singapore, Singapore: IEEE, pp. 1451–1455. Available at: https://doi.org/10.1109/IEEM50564.2021.9673004.

O’Grady, S. (2020) *MongoDB ACID Transactions Whitepaper*, *MongoDB*. Available at: https://www.mongodb.com/resources/products/capabilities/mongodb-multi-document-acid-transactions (Accessed: 16 November 2024).

*Oracle® Fusion Middleware Understanding Oracle TopLink* (2015). Available at: https://docs.oracle.com/middleware/1221/toplink/concepts/app\_dev.htm#OTLCG94288 (Accessed: 27 October 2024).

Patil, M.M. *et al.* (2017) ‘A qualitative analysis of the performance of MongoDB vs MySQL database based on insertion and retriewal operations using a web/android application to explore load balancing — Sharding in MongoDB and its advantages’, in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pp. 325–330. Available at: https://doi.org/10.1109/I-SMAC.2017.8058365.

Pritchett, D. (2008) ‘BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability.’, *Queue*, 6(3), pp. 48–55. Available at: https://doi.org/10.1145/1394127.1394128.

Reinero, B. (2017) *Transitioning from Relational Databases to MongoDB - Data Models | MongoDB Blog*, *MongoDB*. Available at: https://www.mongodb.com/blog/post/transitioning-from-relational-databases-to-mongodb (Accessed: 22 September 2024).

Reisner, P. (1988) ‘Query Languages1’, in M. Helander (ed.) *Handbook of Human-Computer Interaction*. Amsterdam: North-Holland, pp. 257–280. Available at: https://doi.org/10.1016/B978-0-444-70536-5.50017-8.

Salander, A.-K. (2022) *Using atomic counters in the Enhanced DynamoDB AWS SDK for Java 2.x client | AWS Developer Tools Blog*, *AWS Amazon*. Available at: https://aws.amazon.com/blogs/developer/using-atomic-counters-in-the-enhanced-dynamodb-aws-sdk-for-java-2-x-client/ (Accessed: 17 November 2024).

Sen, P.S. and Mukherjee, N. (2024) ‘An ontology-based approach to designing a NoSQL database for semi-structured and unstructured health data’, *Cluster Computing*, 27(1), pp. 959–976. Available at: https://doi.org/10.1007/s10586-023-03995-y.

Seppälä, I. (2024) ‘Java Design Patterns’. Available at: https://github.com/iluwatar/java-design-patterns (Accessed: 17 November 2024).

Shareef, T.H., Sharif, K.H. and Rashid, B.N. (2022) ‘A Survey of Comparison Different Cloud Database Performance: SQL and NoSQL’, *Passer Journal of Basic and Applied Sciences*, 4(1), pp. 45–57. Available at: https://doi.org/10.24271/psr.2022.301247.1104.

Stonebraker, M. (2010) ‘SQL databases v. NoSQL databases’, *Commun. ACM*, 53(4), pp. 10–11. Available at: https://doi.org/10.1145/1721654.1721659.

Stonebraker, M. and Pavlo, A. (2024) ‘What Goes Around Comes Around... And Around...’, in *SIGMOD*. *2024 ACM SIGMOD/PODS Conference*, Santiago, Chile: ACM (2), pp. 21–37. Available at: https://doi.org/10.1145/3685980.3685984.

Sudiartha, I.K.G. *et al.* (2020) ‘Data Structure Comparison Between MySql Relational Database and Firebase Database NoSql on Mobile Based Tourist Tracking Application’, *Journal of Physics: Conference Series*, 1569(3), p. 032092. Available at: https://doi.org/10.1088/1742-6596/1569/3/032092.

Thakare, A. *et al.* (2023) ‘NoSQL Databases: Modern Data Systems for Big Data Analytics - Features, Categorization and Comparison’, *International Journal of Electrical and Computer Engineering Systems*, 14(2), pp. 207–216. Available at: https://doi.org/10.32985/ijeces.14.2.10.

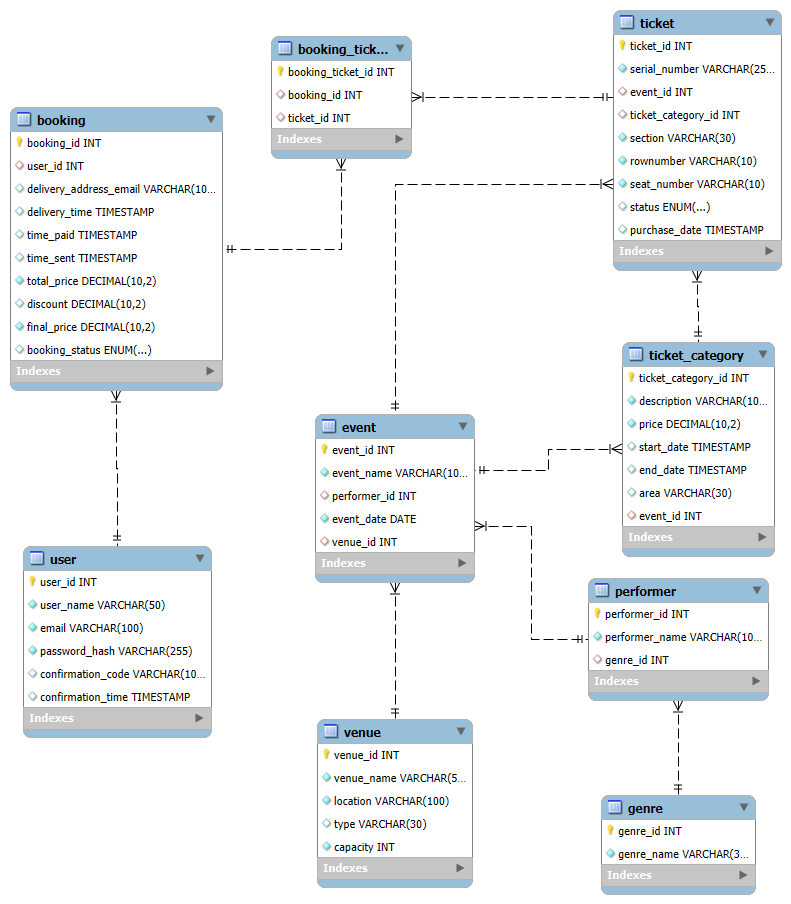
Wodyk, R. and Skublewska-Paszkowska, M. (2020) ‘Performance comparison of relational databases SQL Server, MySQL and PostgreSQL using a web application and the Laravel framework’, *Journal of Computer Sciences Institute*, 17, pp. 358–364. Available at: https://doi.org/10.35784/jcsi.2279.

Yedilkhan, D. *et al.* (2023) ‘Performance Analysis of Scaling NoSQL vs SQL: A Comparative Study of MongoDB, Cassandra, and PostgreSQL’, in *2023 IEEE International Conference on Smart Information Systems and Technologies (SIST)*. *2023 IEEE International Conference on Smart Information Systems and Technologies (SIST)*, pp. 479–483. Available at: https://doi.org/10.1109/SIST58284.2023.10223568.

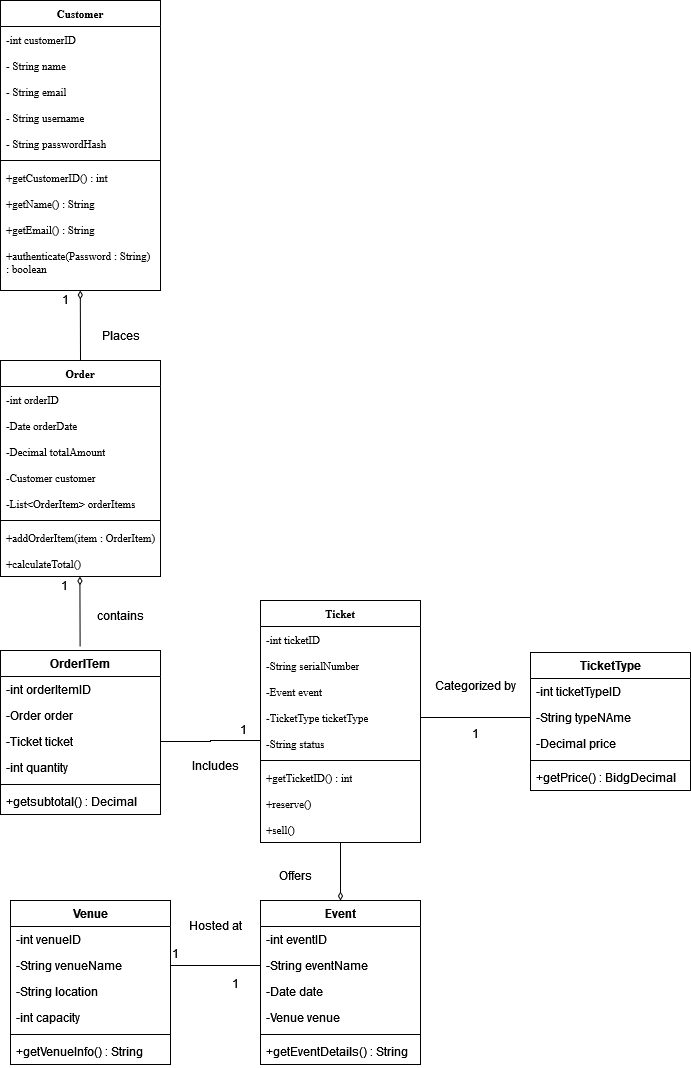
# Appendices

## MySQL Schema Implementation

## Entity-Relationship Diagram



## MySQL Class Diagram



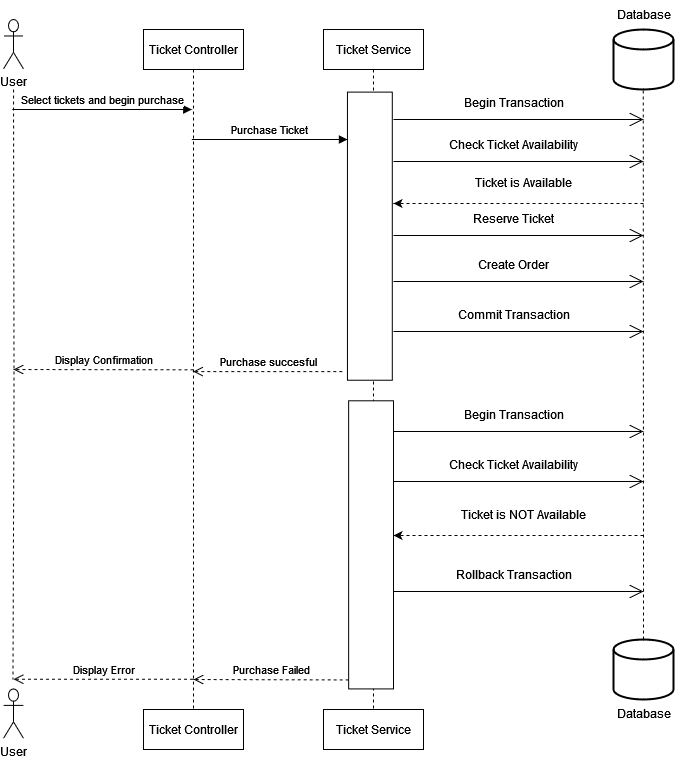
## Document Structure Diagram for MongoDB

## Ticket Purchase process diagram

A diagram of a customer service

Description automatically generated

## Sequence Diagram for Ticket Purchase Transaction



## Backend application diagram

## Deployment with technological stack

## Component Diagram

Depicts the high-level components of the system and their interactions

## MySQL MoSCOW

A table of informational text

Description automatically generated with medium confidence

## MongoDB MoSCOW

A table of informational text

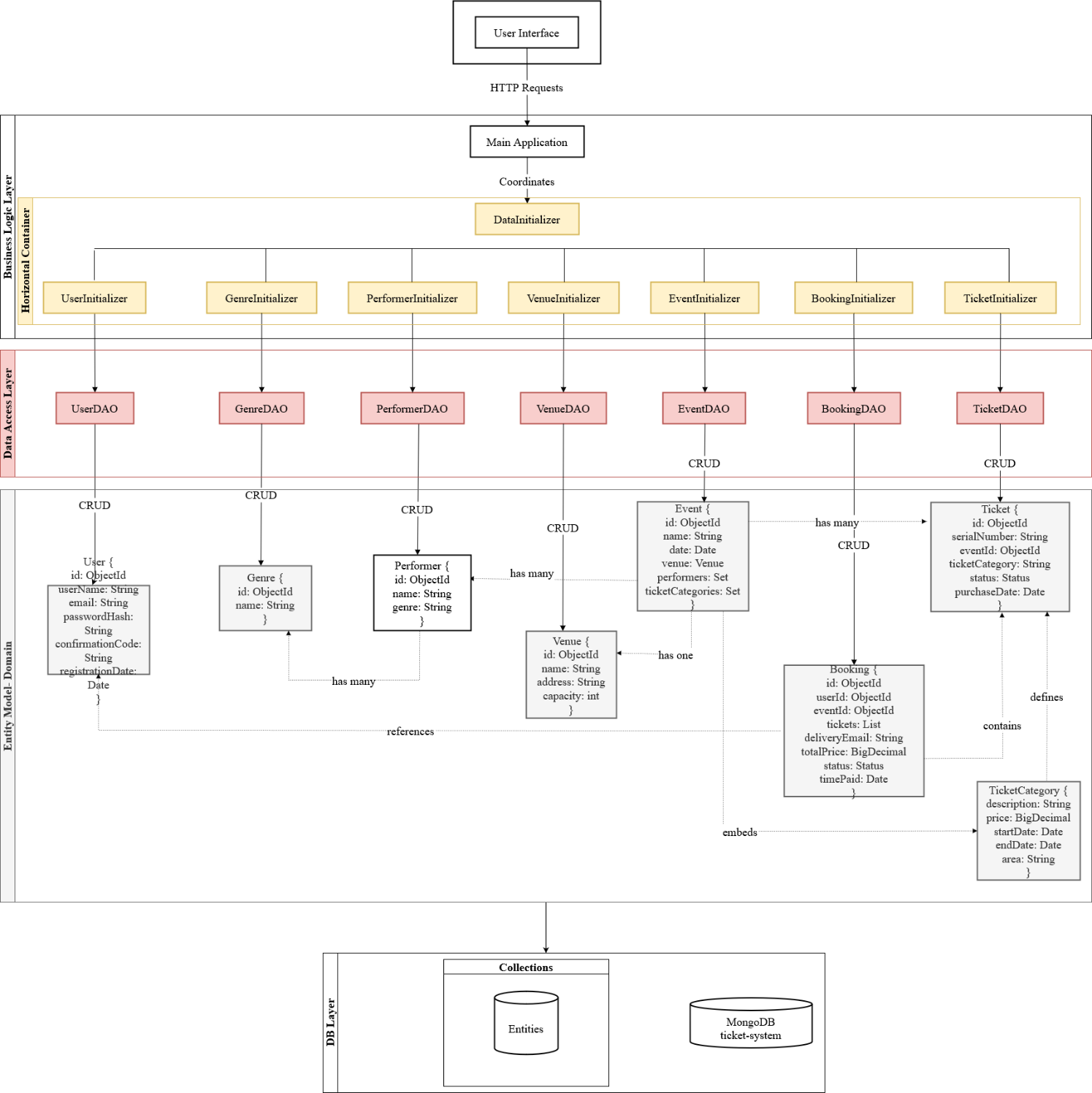
Description automatically generated with medium confidence

## Code samples

## Test results

## Configuration files

## Architecture Diagram for Ticketing System using MongoDB



## Transaction Flow Protocol MongoDB

A close-up of a diagram

Description automatically generated

## Transaction Flow Protocol MongoDB

public class BookingSimulation {

private final BookingService bookingService;

private final UserDAO userDAO;

private final EventDAO eventDAO;

private final TicketDAO ticketDAO;

private final BookingDAO bookingDAO;

private final Datastore datastore;

/\*\*

\* Runs the booking simulation.

\*/

public void runSimulation(ObjectId eventId, int numUsers, int maxTicketsPerUser) {

ExecutorService executor = Executors.newFixedThreadPool(1); // Adjust thread pool size as needed

List<Callable<Boolean>> tasks = new ArrayList<>();

// Retrieve all users to simulate booking attempts

List<User> users = userDAO.findAll();

// If not enough users

if (users.size() < numUsers) {

System.out.println("Not enough users in the system. Please add more users for the simulation.");

executor.shutdown();

return;

}

Random random = new Random();

for (int i = 0; i < numUsers; i++) {

final User user = users.get(random.nextInt(users.size()));

final int ticketsToBook = random.nextInt(maxTicketsPerUser) + 1; // 1 to maxTicketsPerUser

Callable<Boolean> task = () -> {

return bookingService.bookTickets(user.getId(), eventId, ticketsToBook);

};

tasks.add(task);

}

try {

List<Future<Boolean>> results = executor.invokeAll(tasks);

// Wait for all tasks to complete

executor.shutdown();

executor.awaitTermination(1, TimeUnit.MILLISECONDS);

// Output results

System.out.println("Simulation completed.");

System.out.println("Total booking attempts: " + numUsers);

System.out.println("Successful bookings: " + bookingService.getSuccessfulBookings());

System.out.println("Failed bookings: " + bookingService.getFailedBookings());

// Verify no overselling

long totalBookedTickets = bookingDAO.findAll().stream()

.filter(booking -> booking.getEventId().equals(eventId))

.mapToInt(booking -> booking.getTickets().size())

.sum();

long totalTickets = ticketDAO.countAvailableTickets(eventId) + totalBookedTickets;

System.out.println("Total tickets available before booking: " + totalTickets);

System.out.println("Total tickets booked: " + totalBookedTickets);

System.out.println("Tickets remaining: " + ticketDAO.countAvailableTickets(eventId));

} catch (InterruptedException e) {

System.err.println("Simulation interrupted: " + e.getMessage());

e.printStackTrace();

}

}

}