Project Report

ADAD METI ISCTE
Rodrigo Ferreira
No. 140510

https://github.com/rodfer0x80/meti-adad

2025

Abstract

This report details the design and implementation of a distributed web application focused on high data availability and horizontal scalability. A full-stack solution featuring a React frontend, a Node.js backend API, and a sharded MongoDB cluster. The entire infrastructure is containerized using Docker and orchestrated via Docker Compose, utilizing Nginx as a secure reverse proxy. The primary objective is to demonstrate advanced techniques in Storage and Data Availability by implementing a database architecture capable of handling large datasets through horizontal partitioning, sharding, while ensuring service isolation and availability.

Contents

Introduction

1.1 Objectives

The primary objective of this project is to architect and deploy a robust system that addresses the challenges of modern data storage and availability.

- **Scalability:** Implementation of a database layer that supports horizontal scaling through Sharding.
- **Availability:** Ensuring data persistence and access through a distributed architecture.

1.2 Document Organization

This document is structured as follows: Chapter 2 analyzes the requirements and proposes the architecture. Chapter 3 details the logical and physical design of the solution, including database modeling. Chapter 4 describes the implementation details and code structure. Chapter 5 discusses the results and limitations, and Chapter 6 provides conclusions and future work.

Analysis and Planning

2.1 Statement and Requirements

2.1.1 Functional Requirements

- **User Interface:** A web-based frontend to allow users to interact with the system, view and input data.
- **Data API:** A RESTful backend API service to process requests and manage business logic.
- **Distributed Storage:** The system must store data in a non-relational database capable of distributing records across multiple nodes.

2.1.2 Non-Functional Requirements

- **Availability:** The database must ensure data persistence and high availability through the configuration of each shard as a replica set, enabling automatic failover and seamless data redundancy in case of node failure.
- **Scalability:** The system must support horizontal scaling by implementing a sharded cluster. The chosen shard key must facilitate the uniform distribution of data chunks and query load across all defined shards.
- **Performance:** Efficient handling of concurrent requests via the asynchronous I/O model and serviced to the frontend.

2.2 Architecture

2.2.1 Model Description

The solution adopts a microservices-oriented architecture. Unlike monolithic applications, this project decouples the presentation layer, logic layer, and data layer into separate containers. The core innovation lies in the data layer, which utilizes a MongoDB Sharded Cluster rather than a standalone instance.

The architecture ensures that the backend communicates with a Mongos Router, which acts as a query router, abstracting the complexity of the underlying shards and config servers.

2.3 Tech Stack

- **Languages:**
  - **JavaScript:** The primary language for both client and server-side logic [**?**].
  - **Bash:** Used for automation and initialization scripts [9].
- **Frameworks & Runtimes:**
  - **Node.js:** Chosen for its non-blocking I/O event loop, ideal for scalable network applications [1].
  - **Express.js:** A minimal web framework for routing and middleware.
  - **React:** Used for building the component-based user interface [2].
- **Infrastructure:**
  - **Docker:** For containerization of all services [6].
  - **Alpine Linux:** Utilized as the base image for containers to minimize footprint and attack surface [7].
  - **Nginx:** High-performance HTTP server and reverse proxy [5].
  - **MongoDB:** NoSQL database chosen for its native sharding capabilities [3].

Design

3.1 Logical Architecture

The system's logical flow follows a standard web application pattern enhanced by a distributed database:

1. **Client Request:** The user accesses the application via HTTPS. 2. **Reverse Proxy:** Nginx intercepts the request. Static assets are served directly; API requests are forwarded to the Backend. 3. **API Layer:** The Node.js application validates the request. 4. **Data Layer:** The application queries the 'mongos' router. The router consults the Config Servers to identify the correct Shard and retrieves/writes the data.

3.2 Physical Architecture

3.2.1 Topology

The physical architecture is defined by the docker-compose.yaml orchestration. All services reside in a private virtual network.

- **Frontend Container:** Port 3000 (Internal).
- **Backend Container:** Port 5000 (Internal).
- **Database Cluster:**
  - *Config Servers:* Store metadata and chunk distribution.
  - *Shards:* Hold the actual data subsets.
  - *Mongos:* Router (Port 27020 Internal).
- **Nginx:** Ports 80 and 443 exposed to the host system as 8080 and 8443 respectively.

3.2.2 Security

Security is enforced through network isolation. The database is not exposed to the public. Correct implementation and security of the API should prevent abuse and leaking of data.

3.3 Database Modeling

The database utilizes a Document-Oriented model, binary JSON format, BSON.

Performance Consideration: A hashed index or a high-cardinality field is preferred to ensure uniform distribution of chunks across the cluster. This is a valid approach for high throughput environments lacking behind a monolithic approach in a small environment like the one used for testing.

Implementation

4.1 Main Functionalities

The system implements a full CRUD (Create, Read, Update, Delete) cycle. The initialization process is automated: upon container startup, a script detects if the MongoDB cluster is uninitialized and automatically configures the Replica Sets and Shards.

4.2 Structure

- **/backend**: Node.js backend API and npm dependencies
- **/frontend**: React application frontend and npm dependencies
- **/nginx**: Configuration files for the proxy server
- **/scripts**: Python and Bash scripts used for orchestrating the database cluster, backups and other data parsing utilities
- **/database**: MongoDB configuration and data.

Results and Discussion

5.1 Evaluation

The project successfully demonstrates the functional integration of a distributed storage system within a web application. The use of docker allowed for the simulation of a complex infrastructure on a single development machine runnable in any common OS. The sharding mechanism functioned as expected, distributing data logically across the defined shards.

5.2 Limitations

- **Hardware Constraints:** Since all containers run on a single host kernel, the performance benefits of sharding (I/O throughput) are not realized in this demonstration environment.
- **Complexity:** Managing a sharded cluster adds significant operational overhead compared to a monolithic database.

5.3 Objectives

The project fulfills all the project requirements.

Conclusion and Future Work

6.1 Conclusion

The *meti-adad* project serves as a proof-of-concept for building scalable cloud-native applications. It integrates modern JavaScript technologies (React, Node.js) with robust infrastructure tools (Docker, Nginx) and advanced database concepts (MongoDB Sharding).

6.2 Possible Improvements

1. **Authentication:** Adding user authentication to the frontend, backend, database interaction leveraging proper password protection and 2FA along with tokens as cookies to maintain the user temporarily logged in during usage while maximising security for the user's data. It could be implemented by a third party such as JWT or OTP login or manually implemented in the backend.
2. **Backend Scalability:** Better scalability on the nginx server and backend by leveraging multiple routing to different instances of the API backend.
3. **Bot and Automated Traffic Mitigation:** Implement reCAPTCHA and WAF rate limiting to differentiate between legitimate and malicious automated traffic. This is crucial for resource cost reduction, prevention of system over-throttling, and mitigation of DoS attacks and excessive data scraping for AI training.
4. **Data Privacy and Regulatory Compliance:** Establish robust data handling protocols to ensure compliance with international standards. This includes implementing all requirements of the GDPR (e.g., consent management, data portability) and adopting security management best practices based on ISO/IEC 27001.
5. **Application Security:** Integrate security across the full stack. This involves implementing backend measures, frontend measures (e.g., strong Content Security Policy and CSRF tokens), and rigorous security testing.
6. **Features:** More essential features in the frontend to manage the API and data such as event management features.
7. **Monitoring:** Implementing Prometheus and Grafana to visualize the load distribution among shards.
8. **Testing:** Adding tests to the backend API using a framework like jest to ensure the code matches the specs.

9. **Types:** Using a better programming language implementation that leverages type theory to help with code maintainability and development by statically type checking data types in the backend API code. Such could be a slight improvement by using a language like Typescript or by even better using a statically typed compiled language, such as C++, Java or Go, that type checks at compile time instead of transpile time as it is easier, more accurate and faster to perform such checks and balances.

10. **Categories:** Using a language that leverages Category Theory to mathematically verify that the code behaves as expected at compile time, including side effects (e.g. IO), such as Haskell, Scala or Elixir, can ensure easier and faster development as well as maintainability and scalability of the codebase.

Bibliography

[1] Node.js Foundation. *Node.js Documentation*. Available at: https://nodejs.org/en/docs/

[2] Meta Platforms, Inc. *React Documentation*. Available at: https://react.dev/

[3] MongoDB, Inc. *MongoDB Documentation*. Available at: https://www.mongodb.com/docs/

[4] OpenJS Foundation. *Express Documentation*. Available at: https://expressjs.com/

[5] F5, Inc. *NGINX Documentation*. Available at: https://nginx.org/en/docs/

[6] Docker, Inc. *Docker Documentation*. Available at: https://docs.docker.com/

[7] Alpine Linux Development Team. *Alpine Linux*. Available at: https://www.alpinelinux.org/

[8] Arch Linux Community. *Arch Linux Wiki - Docker & System Administration*. Available at: https://wiki.archlinux.org/

[9] Free Software Foundation. *Bash Reference Manual*. GNU Project.

[10] Llamas, Luis. *Luis Llamas - Engineering, Programming and Arduino Tutorials*. Available at: https://www.luisllamas.es/en/

Annexes

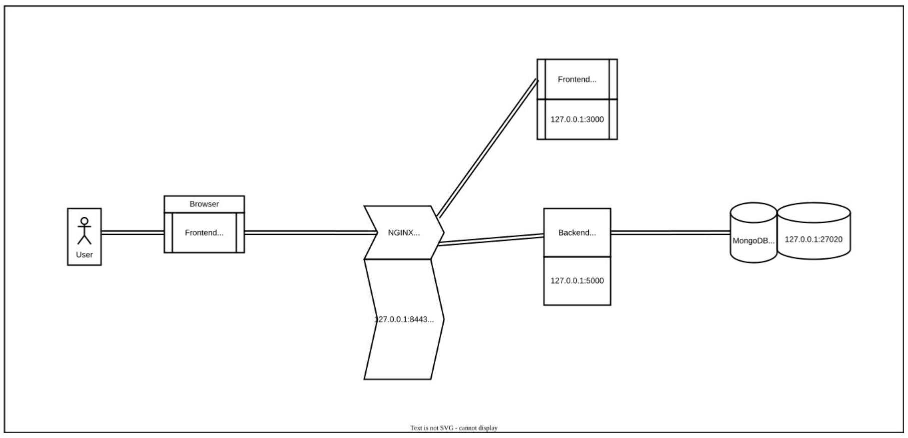

Figure A.1: Architecture Diagram of the ADAD Project.