CipherShare - Decentralized Secure Cloud Storage

C103 Vedant Kothari
C100 Shaurya Patil
C109 Tanishq Nabar
C083 Aniruddha Gurjar

GitHub Link: https://github.com/VedantKothari01/CipherShare

Introduction to the Project

This project focuses on developing a scalable, reliable application using Microservices Architecture combined with optimized and secure database design. The goal is to create a distributed system with independent, loosely coupled services that scale dynamically, ensuring high availability and fault tolerance. Secure communication between services will be established using REST APIs and asynchronous messaging. A significant emphasis will also be placed on designing a database that ensures data integrity, query performance, and security while supporting microservices scalability. Tools like Docker and Kubernetes will enable efficient cloud deployment and management.

Problem Statement

As businesses adopt cloud-based solutions, traditional monolithic applications struggle to meet the growing demands for scalability, fault tolerance, and rapid iteration. This project aims to design a system that efficiently handles large-scale traffic, ensures data security, and maintains performance while also ensuring the database design supports scalability for microservices.

Objectives

The core objectives of this project are:

1. **Microservices Architecture Design:** To design and implement a microservices-based system where each service is autonomous, scalable, and deployable independently. Each microservice will be responsible for a specific business function such as user management, payment processing, etc.
2. **Optimized Database Design:** To create a relational database schema that efficiently handles the application's data needs, including the development of normalized tables, relationships, and indexes for fast querying. The database will ensure high data integrity, security, and support for microservices scalability.
3. **Secure API Communication:** To implement secure communication between microservices using RESTful APIs and, where necessary, asynchronous messaging (such as RabbitMQ or Kafka) to ensure smooth, non-blocking interactions.
4. **Containerization & Orchestration:** To use Docker for containerizing services and Kubernetes for automating the deployment, scaling, and management of containers.

Kubernetes will handle the orchestration, ensuring that the system scales based on demand without compromising performance.

5. **Security:** To ensure robust security by integrating authentication and authorization protocols (OAuth 2.0), encrypting sensitive data, and securing communication channels between services.
6. **Scalability & Performance:** To ensure that the application can scale efficiently by adding or removing services based on demand, while ensuring that data storage and queries remain performant even under high load.

Scope of the Project

1. **Microservices Architecture:** The development of independent, modular services that handle specific functionalities, such as user management, payment processing, and product catalog management. Each service will have its own database schema, allowing for independent scaling and updating.
2. **Database Design:** The relational database design will be optimized for high-performance queries, with normalized tables, indexes, and efficient transaction handling. Additionally, the database will be designed to ensure high availability and fault tolerance, using strategies like replication and sharding where appropriate.
3. **Service Communication:** The microservices will interact via RESTful APIs, and asynchronous communication will be facilitated using RabbitMQ or Kafka to handle tasks that require non-blocking operations or event-driven processing.
4. **Containerization & Orchestration:** The microservices will be containerized using Docker, ensuring that they can run consistently across different environments. Kubernetes will handle service discovery, scaling, load balancing, and the overall management of containers in the deployment environment.
5. **Security Implementation:** OAuth 2.0 will be implemented for user authentication and authorization, ensuring secure access to the services. Data security will be enforced through encryption for both data in transit and data at rest, as well as through secure service-to-service communication.
6. **Testing & Documentation:** Rigorous testing will be conducted to ensure system reliability and performance. The system will undergo functional, integration, and load testing. The project will also include comprehensive documentation on the system architecture, database design, and deployment steps.

Technologies Used

- **Microservices Architecture:**
  - Java/Spring Boot for developing microservices due to its scalability and flexibility in building RESTful APIs.
  - Spring Cloud for service discovery, circuit breakers, and API gateway management.
- **Database Design:**
  - MySQL/PostgreSQL for relational database management, offering high availability, scalability, and support for complex queries.
  - Hibernate ORM for data persistence, ensuring smooth communication between the services and the database.
- **Containerization & Orchestration:**
  - Docker for containerizing the services, providing a consistent environment across all stages of development and deployment.

    - Kubernetes for orchestrating containerized applications, ensuring automated scaling, load balancing, and fault tolerance.
- **API Communication:**
    - RESTful APIs for synchronous communication between microservices.
    - RabbitMQ/Kafka for asynchronous communication to handle non-blocking tasks and enable event-driven architectures.
- **Security:**
    - Spring Security for managing authentication and authorization.
    - OAuth 2.0 for securing APIs and ensuring only authorized users have access to sensitive data and operations.
    - HTTPS for securing communication between clients and services.
- **Deployment:**
    - Jenkins for automating the CI/CD pipeline, ensuring continuous integration, testing, and deployment.
    - Helm for managing Kubernetes deployments via packaged configurations (charts).
- **Testing & Monitoring:**
    - JUnit for unit testing and integration testing of the microservices.
    - Prometheus & Grafana for monitoring service health, resource utilization, and generating alerts for any issues.
    - Postman for API testing and validation of endpoints.
- **Version Control:**
    - Git (GitHub/GitLab) for managing source code and ensuring collaboration across the team.

Expected Outcome

1. **Scalable System**: The system will scale dynamically to handle varying traffic loads while maintaining performance.
2. **Optimized Database**: A high-performance, scalable database with normalized schemas and efficient queries.
3. **High Availability & Fault Tolerance**: Resilient services that automatically restart in case of failure.
4. **Seamless Communication**: Efficient communication between microservices using both synchronous and asynchronous methods.
5. **Security**: Secure APIs and encrypted communication ensuring data protection.
6. **Real-time Monitoring**: Continuous monitoring for system health and resource utilization.
7. **CI/CD Pipeline**: A modern DevOps pipeline to automate testing, deployment, and updates.

ERD Overview

Entities:

1. **Users**
    - userID (PK)
    - username
    - email
    - passwordHash
    - role
    - phoneNumber
    - createdAt
2. **Files**
    - fileID (PK)
    - fileName
    - fileType
    - fileSize
    - encryptedPath
    - description
    - ownerID (FK)
    - createdAt
    - updatedAt
3. **FileVersions**
    - versionID (PK)
    - fileID (FK)
    - versionNumber
    - timestamp
    - changeLog
4. **SharedFiles**
    - shareID (PK)
    - fileID (FK)
    - sharedWithUserID (FK)
    - accessExpiry
    - permissions
    - status
    - createdAt
5. **BlockchainRecords**
    - recordID (PK)
    - fileID (FK)
    - txnHash
    - timestamp
    - actionType

Relationships:

- **users.userID > files.ownerID**
- **files.fileID > fileVersions.fileID**
- **files.fileID > sharedFiles.fileID**

- **users.userID > sharedFiles.sharedWithUserID**
- **files.fileID > blockchainRecords.fileID**

1. users.userID > files.ownerID

- This represents a one-to-many relationship where one user can own multiple files. Each file in the files table has an ownerID which links to the userID in the users table. This establishes which user owns a particular file.

2. files.fileID > fileVersions.fileID

- This is a one-to-many relationship where a file can have multiple versions. The fileID in the files table links to the fileID in the fileVersions table, allowing each file to have associated versions with details like version number, timestamp, and change log.

3. files.fileID > sharedFiles.fileID

- This is a one-to-many relationship where one file can be shared with multiple users. The fileID in the files table links to the fileID in the sharedFiles table, tracking the sharing of files with different users.

4. users.userID > sharedFiles.sharedWithUserID

- This is a one-to-many relationship where one user can receive shared files from multiple users. The userID in the users table links to the sharedWithUserID in the sharedFiles table, indicating who the file is shared with.

5. files.fileID > blockchainRecords.fileID

- This is a one-to-many relationship where a file can have multiple blockchain records. The fileID in the files table links to the fileID in the blockchainRecords table, enabling the logging of file transactions (like uploads, modifications) in the blockchain for immutability and integrity tracking.