Voting System Using Microservices for Secure and Scalable Elections

Mukesh Patel School of Technology, Management & Engineering

Diviyaj Bellare
B024, BTech Computer Engineering

Pranav Kini
B038, BTech Computer Engineering

Rajat Bandekar
B050, BTech Computer Engineering

Abhishek Tripathi
B051, BTech Computer Engineering

1. Introduction to the Project

The project aims to develop a distributed voting system leveraging modern microservices architecture and distributed computing principles. The system will allow users to register, vote for candidates, and view real-time results while ensuring high availability, scalability, and fault tolerance. It will be built using Spring Boot, Eureka, Kafka, PostgreSQL, Docker, and Kubernetes, with a focus on distributed computing to handle high concurrency and large-scale data processing.

2. Problem Statement

Traditional monolithic voting systems face challenges such as:

- **Scalability Issues**: Inability to handle large volumes of concurrent users.
- **Single Point of Failure**: Lack of fault tolerance leading to system downtime.
- **Concurrency Problems**: Risks of duplicate votes or race conditions during high traffic.
- **Latency**: Slow response times due to centralized processing.
- **Complexity in Maintenance**: Difficulties in scaling, updating, and debugging monolithic systems.

This project addresses these challenges by designing a distributed, event-driven microservices architecture that ensures scalability, fault tolerance, and eventual consistency.

3. Objectives

- **Design a Distributed System**: Build a **scalable and fault-tolerant** voting system using **microservices**.
- **Ensure High Availability**: Use **Kubernetes** for orchestration and **auto-scaling** to handle traffic spikes.
- **Handle Concurrency**: Implement **distributed locking (Redis)** and **optimistic locking** to prevent duplicate votes.
- **Achieve Eventual Consistency**: Use **Kafka** for asynchronous communication and **Redis** for caching.
- **Provide Real-Time Results**: Aggregate voting results in real-time using **Kafka and Redis**.
- **Simplify Deployment**: Containerize services using **Docker** and orchestrate them using **Kubernetes**.

4. Scope of the Project

The project will focus on the following key functionalities:

- **User Management**: Registration, authentication, and authorization.
- **Candidate Management**: Adding and retrieving candidate details.
- **Voting Mechanism**: Secure and concurrent voting using **Kafka and Redis**.
- **Result Aggregation**: Real-time vote counting and result display.
- **Notifications**: Sending notifications (email/SMS) for successful votes.
- **API Gateway**: Centralized request routing and rate limiting.
- **Service Discovery**: Dynamic service registration and load balancing using **Eureka**.
- **Deployment**: Containerization with **Docker** and orchestration with **Kubernetes**.

5. Technologies Used

Frontend:

- **Flutter**

Backend:

- **Spring Boot** (Microservices)
- **Spring Cloud Gateway** (API Gateway)
- **Spring Cloud Eureka** (Service Discovery)
- **Spring Security + JWT** (Authentication)
- **Kafka** (Event Streaming)
- **Redis** (Caching and Distributed Locking)
- **PostgreSQL** (Database)

DevOps:

- **Docker** (Containerization)
- **Kubernetes** (Orchestration)

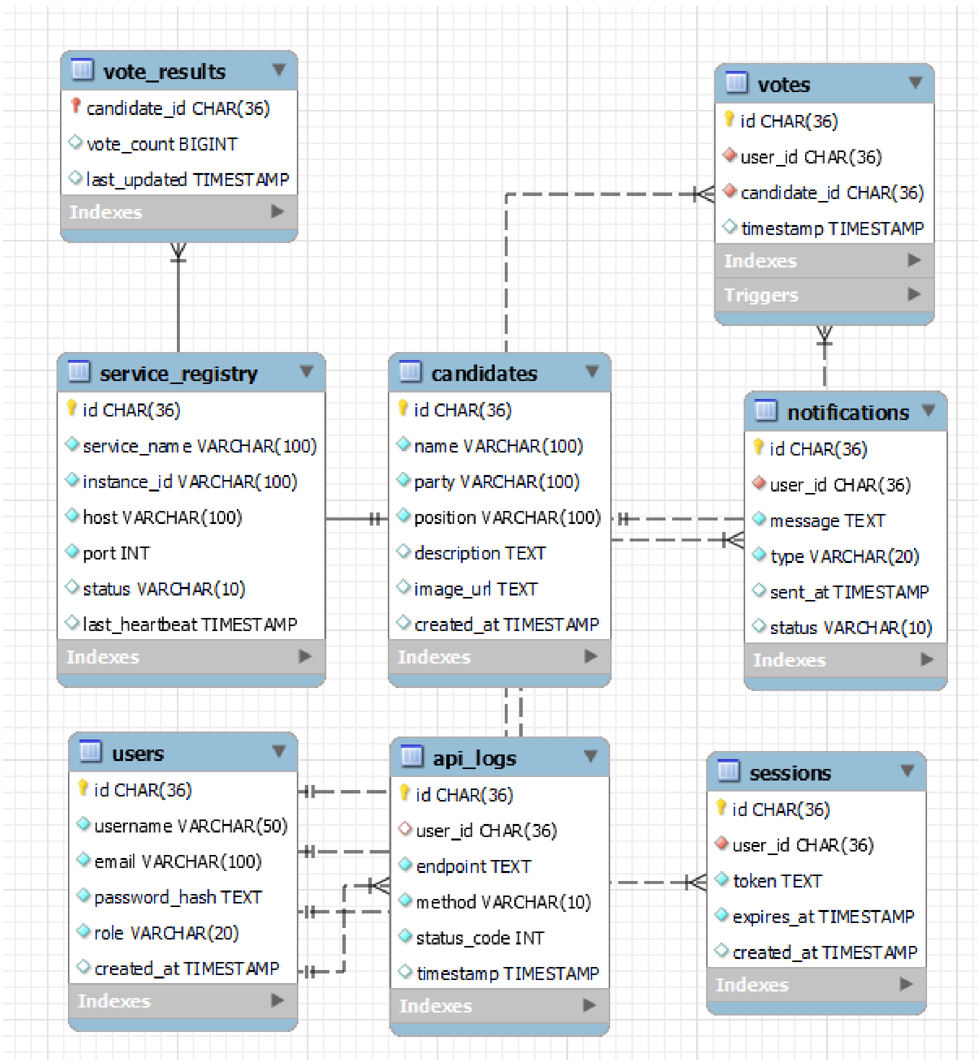
Distributed Computing:

- **Kafka** (Asynchronous Communication)
- **Redis** (Distributed Locks and Caching)
- **Kubernetes** (Auto-scaling and Load Balancing)
- **PostgreSQL Read Replicas** (Database Scalability)

6. Expected Outcome

- **Scalable System**: The system will handle high traffic and scale dynamically using **Kubernetes**.
- **Fault Tolerance**: The system will remain operational even if some components fail, thanks to **Kubernetes and distributed architecture**.
- **Real-Time Results**: Users will see real-time voting results with **eventual consistency**.
- **Secure Voting**: The system will prevent duplicate votes and ensure **data integrity** using **Redis locks and optimistic locking**.
- **Ease of Deployment**: The system will be **easy to deploy and manage** using **Docker and Kubernetes**.
- **Modular and Maintainable**: The **microservices architecture** will make the system modular, easy to update, and maintain.

7. Normalised Entity-Relation Diagram



8. Entity Relationships (ER)

Users Table (Voters & Admins)

- Each user (VOTER or ADMIN) can have multiple sessions.
- VOTER users can vote once in each election.
- ADMIN users manage candidates and view results.

Candidates Table

- Candidates belong to a political party.
- Users vote for candidates.
- Votes update results asynchronously.

Votes Table

- Each vote belongs to one user and one candidate.
- A user can vote only once (enforced by unique_vote constraint).

Vote Results Table

- Stores the aggregated vote count for each candidate.
- Updates asynchronously via Kafka and triggers.

Notifications Table

- Each vote triggers a notification to the user.
- Notifications are queued and sent asynchronously.

API Logs Table

- Logs API calls for audit and security.
- Tracks who accessed what and when.

Service Registry Table

- Stores active microservice instances for service discovery.

Normalization

First Normal Form (1NF)

- All tables store **atomic** values (no multivalued attributes).
- No repeating columns; relationships are represented through foreign keys.

Second Normal Form (2NF)

- No **partial dependencies** (each non-key attribute depends entirely on the primary key).
- Example: The votes table references both user_id and candidate_id but does not store unnecessary details like usernames or candidate names.

Third Normal Form (3NF)

- No **transitive dependencies** (non-key attributes depend only on the primary key).
- Example: The users table only stores user-specific details, while authentication details like session tokens are in a separate sessions table.

Boyce-Codd Normal Form (BCNF)

- **Every determinant is a candidate key** : In each table, any attribute that determines another must itself be a candidate key.
- **No partial dependencies** : Each non-key attribute fully depends on the entire primary key.
- **No transitive dependencies** : Attributes only depend on the primary key, not on other non-key attributes.
- **Example:** In the sessions table, id is the primary key, and all attributes (like user_id, token, and expires_at) depend only on id, ensuring there are no hidden dependencies.
- **Example:** In the votes table, id is the primary key, and user_id and candidate_id are foreign keys, ensuring votes are correctly referenced without redundant data.

Table	Primary Key	Foreign Keys	Relationships
users	id	-	One-to-Many with sessions, One-to-Many with votes, One-to-Many with notifications
sessions	id	user_id → users(id)	Many-to-One with users
candidates	id	-	One-to-Many with votes, One-to-One with vote_results
votes	id	user_id → users(id), candidate_id → candidates(id)	Many-to-One with users, Many-to-One with candidates
vote_results	candidate_id	candidate_id → candidates(id)	One-to-One with candidates
notifications	id	user_id → users(id)	Many-to-One with users
api_logs	id	user_id → users(id)	Many-to-One with users (nullable)
service_registry	id	-	Stores microservice instances