Scalable Containerized Voting Application

Executive Summary

The objective of this project is to design and implement a scalable, secure, and reliable online voting application capable of handling tens of thousands of concurrent users. The application will be containerized to ensure scalability and ease of deployment. The system will consist of a user-friendly front-end interface, a robust authentication service, a high-performance caching layer, dedicated worker processes for handling data processing, and a secure database for persistent storage.

Project Description

The project allows users to access the different REST endpoints through a reverse proxy in the form of a load balancer. Through the load balancer, a login portal for an admin frontend web page and a user frontend web page can be accessed.
These login portals are able to authenticate and authorize users through a service that securely receives login credentials and generates JSON Web Token (JWT) for use throughout the system.

When an admin logs into the web portal, they receive a JWT. With that JWT they are able to access the admin portal and use the different API endpoints that are allocated for admins. One of the main endpoints allows them to create users in the system.

When a user logs into the web portal, they also receive a JWT token, but this token will deny them access to the admin portal. It will allow them to login to the user portal and vote.
When a user makes a vote, if the JWT is valid, the main Spring Boot API allows access to the /writeToRedis method. The vote is added to the Redis cache where it is temporarily stored.

The worker node is a backend service that reads data from the Redis cache and stores it in PostgreSQL. The purpose of the Redis cache and worker node is to protect the PostgreSQL database from getting overwhelmed in periods of high traffic.

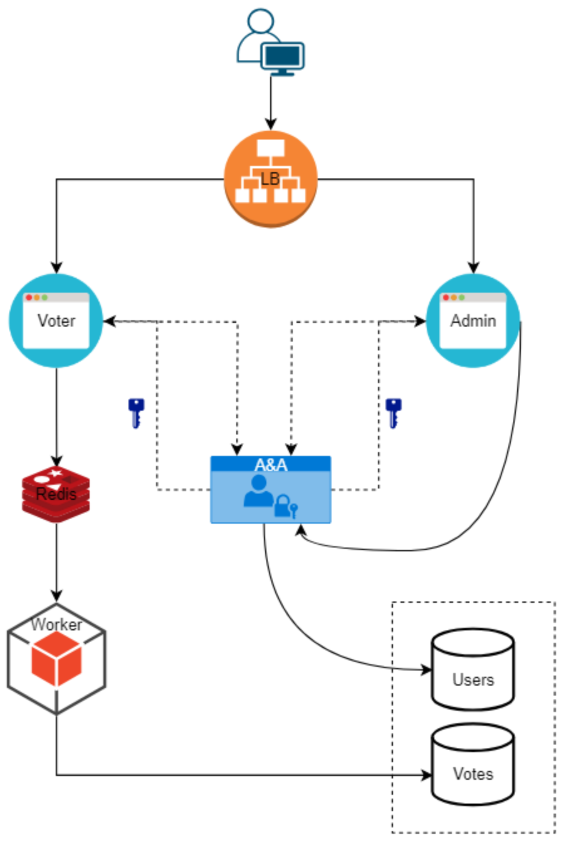
The voting application was developed as a microservice architecture pattern. One could make a strong argument that the path from the user frontend to the Redis to the Worker to the PosgreSQL database is a pipeline. Each service was developed with containers in mind. Google's JIB dependency was used to automatically create containers from Java code at build time. This approach allows for independent scaling and updating of each microservice, leading to better resource management, higher availability, and the ability to scale horizontally very fast with Kubernetes.

Components of the Design

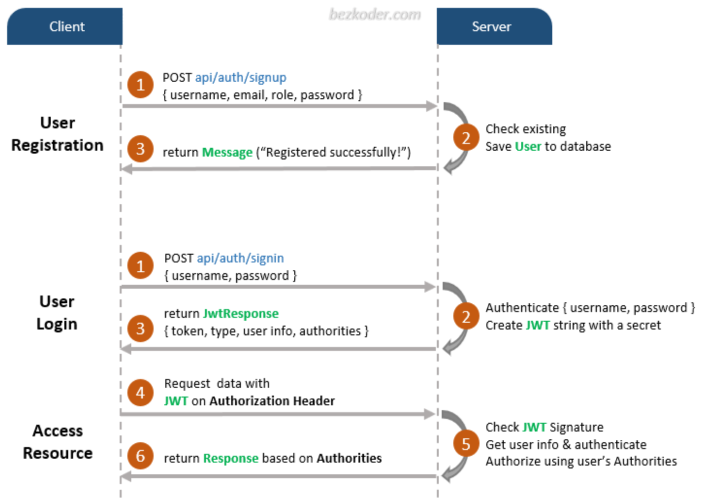
- **Load Balancer (LB):** Distributes incoming traffic across multiple instances of the front end applications, ensuring even load distribution and high availability. It also routes traffic to the correct frontend for admins and users.
- **Front End - User Interface:** This is the front-end component that voters will interact with directly. It is designed to accept user credentials and provide a page to cast votes.
- **Front End - Admin Interface:** This is the front-end component that admins interact with directly. It is designed to accept user credentials and provide a page for admins to add users.
- **Main Voting (API):** Server-side logic, that allows users to write to the Redis cache with the correct JWT.
- **Cache:** Temporarily stores votes to improve performance. It acts as a buffer to reduce the load on the worker processes and the database.
- **Worker:** A backend service that reads from the Cache and writes to the Database. It ensures data integrity and consistency, even under heavy load.
- **Database (DB):** Stores all application data, including user credentials, votes, and election results.
- **Authentication Service (AS):** Handles user authentication, ensuring that each vote is cast by a legitimate voter and admins have the correct privileges.

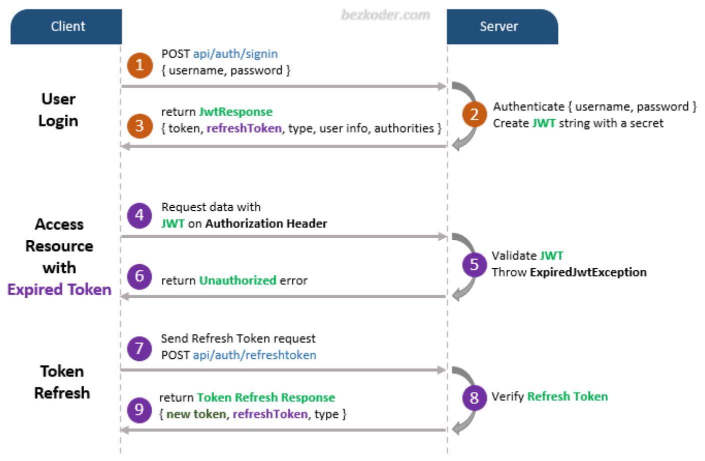
Main Architecture Diagram

The following diagram illustrates the architecture of the containerized voting application:



Sequence Diagrams for Authentication and Authorization


Client
bezkoder.com
Server
User Registration
1 POST api/auth/signup { username, email, role, password }
2 Check existing Save User to database
3 return Message ("Registered successfully!")
User Login
1 POST api/auth/signin { username, password }
2 Authenticate { username, password } Create JWT string with a secret
3 return JwtResponse { token, type, user info, authorities }
Access Resource
4 Request data with JWT on Authorization Header
5 Check JWT Signature Get user info & authenticate Authorize using user's Authorities
6 return Response based on Authorities


Client
bezkoder.com
Server
User Login
1 POST api/auth/signin { username, password }
2 Authenticate { username, password } Create JWT string with a secret
3 return JwtResponse { token, refreshToken, type, user info, authorities }
Access Resource with Expired Token
4 Request data with JWT on Authorization Header
5 Validate JWT Throw ExpiredJwtException
6 return Unauthorized error
Token Refresh
7 Send Refresh Token request POST api/auth/refreshtoken
8 Verify Refresh Token
9 return Token Refresh Response { new token, refreshToken, type }

Architecture Design Records
See end of paper. ADRs were done with adr-tools and pandoc.

Conclusion
The proposed containerized voting application was ultimately created successfully. All of the core components that were proposed are working correctly. I used this project to learn more about Spring Boot and authentication and authorization services. In my day job, I use these authentication and authorization, but they are a black box. I learned a lot about Spring Boot and building secure services.

1. Containerized Voting App

Date: 2023-12-07

Status

Accepted

Context

A voting application is needed to handle high volumes of concurrent users all over the nation.

Decision

A microservice architecture will be used for distributing containers for easy scalability

Consequences

The architecture introduces complexity in terms of setup, management, and monitoring.

2. Docker for Container Development

Date: 2023-12-07

Status

Accepted

Context

A container technology is needed for developing the voting application containers.

Decision

Docker will be used for developing containers.

Consequences

Relying on Docker could be risky if the technology falls out of favor or encounters significant issues in the future.

3. Kubernetes for Deployment

Date: 2023-12-07

Status

Accepted

Context

An orchestration tool is needed to deploy these containers to production.

Decision

Kubernetes will be used as the orchestration tool for distributed computing and deployment of pods.

Consequences

Kubernetes is complex to set up and manage, requiring a steep learning curve and significant expertise.

4. Cache for Concurrent Requests

Date: 2023-12-07

Status

Accepted

Context

As the voting front end applications scale, caches will be needed to prevent the databases from getting overwhelmed.

Decision

Redis will be used for this caching technology.

Consequences

Redis, being an in-memory database, may require significant memory resources, especially under heavy load.

5. Application Security

Date: 2023-12-07

Status

Accepted

Context

A voting application must be secure. Security must be built in from the beginning.

Decision

A dedicated authentication and authorization service using Spring Boot will be built.

Consequences

Spring Boot Security can be complex, requiring a thorough understanding of security concepts and careful implementation to avoid security flaws.

6. Databases for User and Vote Storage

Date: 2023-12-07

Status

Accepted

Context

Databases are needed for storing users and the user's votes.

Decision

PostgreSQL will be used for the database technology.

Consequences

PostgreSQL be more resource-intensive than some other databases, particularly under heavy loads.