Ctrl+Edu - Distributed Educational Campus Management System

Github URL : https://github.com/Naveen-Jaisankar/CtrlEdu

Team Group Name: Jobizzz
Team Members:

- **Naveen Jaisankar 23200145**
- **Shaline Raghupathy 23201217**
- **Nithishh Saravanan 23200138**

Synopsis

Ctrl+Edu is a comprehensive distributed educational campus management system designed to facilitate efficient campus operations for students, lecturers, and super admins. Built with a role-based authentication structure, the system addresses key distributed system challenges such as fault tolerance, scalability, and real-time communication.

Key Features:

- **Role-Based Access Control:** Secure access with Keycloak for students, lecturers, and super admins.
- **Course & Enrollment Management:** The super admin can manage academic configurations, including adding courses, enrolling students, and assigning lecturers.
- **Real-Time Chat Service:** A chat service powered by WebSockets and Kafka allows real-time communication for subject-specific collaboration.
- **Scalability and Fault Tolerance:** The system ensures high availability using technologies like Redis, Kafka, RabbitMQ, and Eureka for service discovery and load balancing.
- **Notification Service:** RabbitMQ-based notification service sends automated emails for critical events such as successful student enrollment or teacher assignment.
- **Gateway Service with Eureka Integration:** The **Gateway Service** uses **Netflix Eureka** for dynamic service discovery, load balancing, and centralized security management, ensuring efficient request handling and API routing.

Overall, Ctrl+Edu is a scalable and fault-tolerant system designed to support the dynamic requirements of an educational institution, ensuring secure data handling, real-time communication, and streamlined administrative processes.

Technology Stack

The Ctrl+Edu project utilizes a comprehensive technology stack tailored for a distributed system, ensuring scalability, fault tolerance, and efficient service communication. The stack is divided into **frontend**, **backend**, and **infrastructure** components.

Frontend:

The frontend is developed using modern technologies to ensure a fast, interactive, and user-friendly experience.

- **React with TypeScript:** React provides a component-based architecture for building a dynamic user interface, while TypeScript enhances code quality with static typing.
- **Vite:** A modern build tool that offers faster development builds and optimized production builds compared to Webpack.
- **Tailwind CSS:** A utility-first CSS framework used for styling and maintaining a consistent design across the application.
- **WebSocket Integration:** Enables real-time communication for the chat feature, allowing students and lecturers to collaborate seamlessly.
- **Keycloak JavaScript Adapter:** Used for handling role-based authentication and session management on the client side.

Why These Technologies?

- **Scalable:** Component-based design simplifies the addition of new features.
- **Performance:** Vite ensures faster build times and optimized production bundling.
- **Security:** Keycloak ensures secure and centralized authentication management.

Backend:

The backend is designed using a microservices architecture for modularity, fault tolerance, and easy service scaling.

Programming Languages & Frameworks:

- **Java:** The primary programming language for the backend services.
- **Spring Boot:** A microservices framework that simplifies the development and orchestration of individual services.

Services & Components:

- **AuthService:** Manages user authentication and authorization using **Keycloak** integration.
- **ChatService:** Real-time messaging service using **Kafka** and **WebSocket**.
- **CommonService:** Provides shared utilities like data transformation and error handling across microservices.
- **EurekaService:** Service discovery and registration using **Netflix Eureka**.
- **GatewayService:** API Gateway powered by **Spring Cloud Gateway** for secure and efficient request routing.

- **MasterService:** Core business logic, including course and student management.
- **NotificationService:** Handles notifications using **RabbitMQ** for message queuing and **Spring Boot Mail** for email dispatch.

Security & Authentication:

- **Keycloak:** Centralized identity provider for role-based authentication and JWT token management.

Infrastructure Components:

Key infrastructure components support fault tolerance, data management, and real-time messaging across services.

- **Kafka:** Distributed event streaming platform used for handling chat messages and real-time data propagation.
- **Zookeeper:** Manages Kafka's distributed coordination for message streaming and partitioning.
- **Redis:** In-memory data store used for caching and enhancing fault tolerance.
- **RabbitMQ:** Message broker for reliable notification delivery (e.g., email notifications).
- **PostgreSQL:** Relational database for storing core application data such as user accounts, courses, and modules.
- **Docker:** Used for containerizing services for consistent deployment across environments.
- **Netflix Eureka:** Service discovery and load balancing for scaling multiple instances of each microservice.
- **Keycloak:** A centralized identity and access management tool for handling authentication and role-based access control. It manages user roles (student, teacher, super-admin) and issues secure **JWT tokens** for service authorization, ensuring consistent security across all microservices.

Why This Stack Was Chosen?

- **Scalability:** Kafka, RabbitMQ, and Eureka ensure services can scale horizontally.
- **Fault Tolerance:** Redis and RabbitMQ provide data persistence and failover mechanisms.
- **Security:** Keycloak ensures centralized, secure authentication management.
- **Performance:** React and Vite deliver optimized frontend performance.
- **Modular Architecture:** Spring Boot and microservices make the system easier to extend and maintain.

System Overview (Hierarchical Structure)

Ctrl+Edu (Distributed Educational Management System)

→ Frontend (React + TypeScript)

  → User Interface Components

  → WebSocket Integration (for Chat)

  → Keycloak Adapter (for Authentication)

  → API Communication (Axios)

→ Backend (Spring Boot)

  → AuthService (Keycloak Integration)

    → JWT Token Management

    → User Role Verification

  → ChatService (WebSocket + Kafka)

    → WebSocket Listener

    → Kafka Producer & Consumer

    → Redis Cache for Message Storage

  → MasterService (PostgreSQL)

    → Course Management

    → Student Enrollment

    → Data Persistence

  → NotificationService (RabbitMQ)

    → Message Queue for Notifications

    → Email Dispatch via SMTP

  → CommonService

    → Shared Utilities and Error Handling

  → GatewayService (Spring Cloud Gateway)

    → API Routing

    → Token Validation

    → Rate Limiting

  → EurekaService (Netflix Eureka)

    → Service Registration

    → Load Balancing & Discovery

→ Infrastructure & Tools

→ Kafka (Real-Time Messaging)

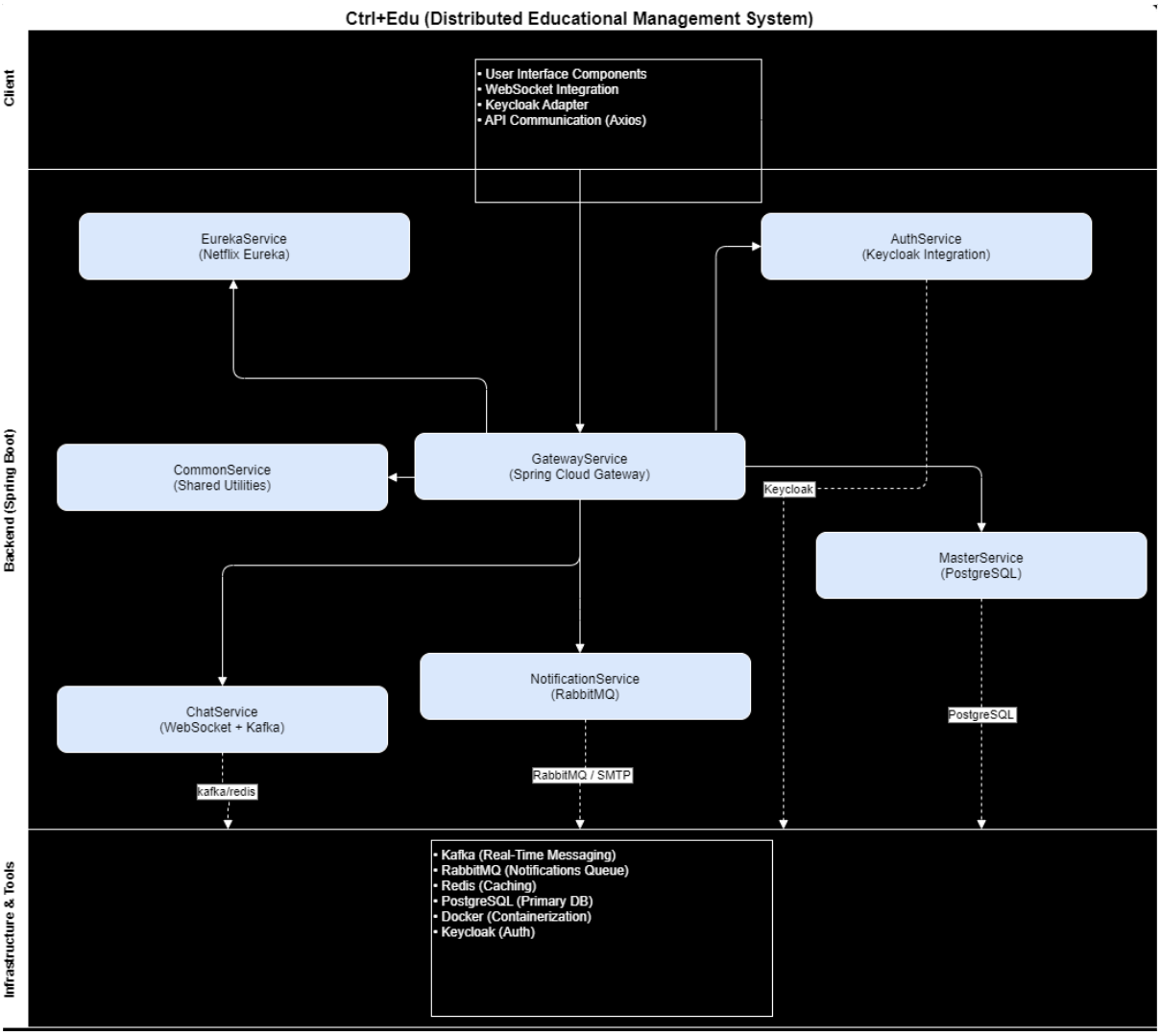
→ RabbitMQ (Message Queue for Notifications)

→ Redis (Caching Layer)

→ PostgreSQL (Primary Database)

→ Docker (Containerization)

→ Keycloak (Centralized Authentication)



Architecture Explanation:

1. Frontend (React + TypeScript)

The frontend is built using **React** with **TypeScript** and is structured as a dynamic, component-driven single-page application (SPA).

Key Components:

- **User Interface Components:** The frontend consists of reusable UI components for login, course management, chat, and notifications.
- **WebSocket Integration:** WebSockets are integrated using React hooks for real-time chat communication with the ChatService.
- **Keycloak Adapter:** The frontend uses the Keycloak JavaScript adapter for secure authentication and role-based authorization.
- **API Communication:** Axios is used for RESTful API communication between the frontend and the GatewayService.

How it Works:

- When a user logs in, the **Keycloak adapter** retrieves and manages the JWT token.
- API requests are sent to the **GatewayService**, which validates the token before routing them to the appropriate backend services.
- WebSocket connections are initiated for real-time chat, and messages are sent directly to the ChatService.

---

2. Backend (Spring Boot Microservices)

The backend is implemented using **Spring Boot**, with each service handling a specific aspect of the application's business logic and operations. Each service runs independently and communicates with others through REST APIs, **Kafka**, and **RabbitMQ**.

AuthService (Keycloak Integration)

The AuthService is responsible for handling user authentication and role-based access control using **Keycloak**.

- **JWT Token Management:** Generates and validates JWT tokens for user sessions.
- **User Role Verification:** Verifies roles such as student, teacher, and super-admin before authorizing actions.

Flow:

1. The frontend initiates a login request.
2. The AuthService validates credentials through **Keycloak**.
3. If successful, a JWT token is generated and returned to the frontend.
4. Subsequent requests from the frontend include the token for authorization.

ChatService (WebSocket + Kafka)

The ChatService is responsible for real-time messaging between students and lecturers using **WebSockets** and **Kafka** for event-driven messaging.

- **WebSocket Listener:** Listens for incoming WebSocket connections and broadcasts messages to connected users.
- **Kafka Producer & Consumer:** Messages are published to **Kafka** topics for persistence and delivered to subscribers.
- **Redis Cache:** Used for storing recent chat messages to improve response times and prevent data loss during failures.

Flow:

1. The frontend opens a WebSocket connection with the ChatService.
2. Messages are sent via WebSocket and published to a **Kafka** topic.
3. The **Kafka Consumer** processes the message and stores it in **Redis** for temporary storage.
4. Messages can be persisted to **PostgreSQL** for long-term storage if needed.

MasterService (PostgreSQL)

The MasterService handles core academic functionalities, such as course management and student enrollment, while ensuring data persistence.

- **Course Management:** The super admin can create, update, and delete courses.
- **Student Enrollment:** Manages student enrollment and class assignments.
- **Data Persistence:** Uses **PostgreSQL** as the primary database for storing course and enrollment data.

Flow:

1. Super admins create a course through the frontend.
2. The **GatewayService** routes the request to MasterService.
3. The service validates the request, stores the data in **PostgreSQL**, and triggers a notification through **RabbitMQ**.

NotificationService (RabbitMQ)

The NotificationService handles all email notifications using **RabbitMQ** and **SMTP**.

- **Message Queue:** Uses **RabbitMQ** to queue notification messages for processing.
- **Email Dispatch:** Emails are sent using **SMTP** when a student is enrolled or a teacher is assigned to a course.

Flow:

1. The MasterService publishes a notification event to **RabbitMQ**.
2. The NotificationService listens for these messages.

3. The message is processed, and an email is sent via **SMTP**.

CommonService

The CommonService provides shared functionalities and utility methods for all other services.

- **Shared Utilities:** Contains reusable code like DTOs, error handling, and data transformation utilities.
- **Error Handling:** Provides a centralized error handling mechanism.

GatewayService (Spring Cloud Gateway)

The GatewayService acts as a centralized API gateway for routing incoming HTTP requests to the appropriate microservices.

- **Token Validation:** Intercepts incoming requests and validates JWT tokens using the AuthService.
- **API Routing:** Routes requests to the relevant backend services.
- **Rate Limiting:** Prevents misuse of the API through request rate limiting.

EurekaService (Netflix Eureka)

The EurekaService is used for service discovery and load balancing in the microservices architecture.

- **Service Registration:** Each microservice registers itself with the **Eureka Server**.
- **Load Balancing:** Distributes incoming traffic across multiple instances of a service.

Flow:

1. All services register themselves with EurekaService.
2. The GatewayService queries **Eureka** to locate the desired service instance.
3. Requests are distributed among multiple instances for load balancing.

---

3. Infrastructure & Tools:

The project relies on the following infrastructure components to ensure fault tolerance and scalability:

- **Kafka:** Used for event-driven message streaming and asynchronous processing in the chat service.
- **RabbitMQ:** Handles message queuing for the NotificationService.
- **Redis:** In-memory caching for chat messages and session management.
- **PostgreSQL:** Relational database for persistent storage.

- **Docker:** Used for containerizing microservices and ensuring consistent deployment.
- **Keycloak:** Manages authentication and role-based authorization.
- **Netflix Eureka:** Provides service discovery and load balancing across microservices.

System Flow Summary:

1. **User Authentication:**
    - Users log in via the frontend using Keycloak.
    - Keycloak issues a JWT token that is validated by AuthService.
2. **Course Management:**
    - The super admin adds a course via the frontend.
    - The GatewayService routes the request to the MasterService.
    - The course is stored in **PostgreSQL**, and a notification event is sent to NotificationService.
3. **Real-Time Chat:**
    - Users join a chat group through the frontend.
    - The ChatService receives the message through **WebSockets**.
    - Messages are published to **Kafka** and stored in **Redis** for temporary storage.
4. **Notification Handling:**
    - MasterService triggers notifications through **RabbitMQ**.
    - NotificationService consumes the message and sends an email via **SMTP**.

---

1. Scalability:

Scalability refers to the ability of the system to handle increasing workloads by either expanding resources (vertical scaling) or adding more instances (horizontal scaling).

Key Strategies for Scalability:

- **Microservices Architecture:**
    - Each service (AuthService, ChatService, MasterService, etc.) runs independently and can be scaled individually based on demand.
    - Example: If chat traffic increases, only the ChatService can be scaled without affecting other services.
- **Kafka for Event Streaming:**
    - **Kafka** handles distributed message streaming and can manage large volumes of real-time chat messages.

    - **Partitioning:** Kafka partitions data streams to distribute message handling across multiple brokers.

- **RabbitMQ for Notification Queuing:**
    - **RabbitMQ** ensures notification messages are queued for asynchronous processing.
    - If the NotificationService experiences high traffic, RabbitMQ can hold messages while scaling up the service.
- **Service Discovery with Netflix Eureka:**
    - **EurekaService** enables service registration and dynamic discovery of available service instances.
    - It allows **GatewayService** to distribute incoming requests among multiple instances of a service (load balancing).
- **Horizontal Scaling via Docker and Kubernetes:**
    - **Docker** containerizes each microservice, making it easy to deploy multiple instances.
    - **Kubernetes** (if implemented) can auto-scale based on traffic patterns.

---

2. Fault Tolerance:

Fault tolerance ensures that the system remains operational even when parts of it fail.

Key Strategies for Fault Tolerance:

- **Service Redundancy:**
    - Multiple instances of each microservice are deployed using **EurekaService**, ensuring continued service availability even if one instance fails.
- **Redis for Caching and Failover Management:**
    - **Redis** is used for:
        - Caching frequently accessed data (like recent chat messages).
        - Temporary storage in case of database failures to avoid message loss.
    - If the primary database is temporarily unreachable, **Redis** ensures data availability.
- **Kafka for Message Durability:**
    - **Kafka** retains messages for a configurable duration, ensuring no data loss during temporary service failures.
    - Messages are stored in a fault-tolerant, distributed cluster with replication across multiple brokers.
- **RabbitMQ for Guaranteed Delivery:**
    - **RabbitMQ** queues ensure reliable delivery of notification messages even during service downtimes.

    - Messages are persisted in the queue until successfully processed.
- **Keycloak for High-Availability Authentication:**
    - **Keycloak** supports clustered deployments for failover authentication.
    - In case of a node failure, other instances handle the load without service interruption.

---

3. Load Balancing and Traffic Management:

- **Netflix Eureka:** Manages service discovery and dynamic load balancing across instances.
- **Spring Cloud Gateway:**
    - Handles traffic routing and applies rate-limiting strategies to prevent service overload.
    - Token validation occurs at the gateway level, reducing redundant checks within services.

---

4. Disaster Recovery and Backup:

- **PostgreSQL:** Configured with database replication and scheduled backups for data protection.
- **Kafka & RabbitMQ:** Implement persistent message storage, ensuring recovery during unexpected failures.
- **Docker Containers:** Enable fast restoration of services from pre-built images in case of server crashes.

Summary of Scalability and Fault Tolerance Measures:

Component	Scalability Feature	Fault Tolerance Feature
AuthService	Horizontally scalable Keycloak cluster.	Keycloak replication ensures failover.

ChatService	Kafka partitions for parallel processing.	Redis caching and Kafka message durability.
NotificationService	RabbitMQ queues for asynchronous scaling.	RabbitMQ stores undelivered messages.
GatewayService	Eureka for dynamic routing and scaling.	Load balancing across multiple instances.
MasterService	Multiple PostgreSQL instances for read scaling.	PostgreSQL replication and backup strategies.
EurekaService	Multi-node deployment for load balancing.	Failover by multiple registry instances.

Conclusion:

The **Ctrl+Edu** system effectively handles scalability through a microservices architecture, event-driven messaging, and service discovery mechanisms. Fault tolerance is reinforced using caching strategies, message brokers, and service redundancy, making the system resilient and capable of handling failure scenarios without significant disruption.

Contributions

The **Ctrl+Edu** project was collaboratively developed by three team members, with each focusing on specific services and ensuring smooth integration across the system.

Naveen Jaisankar worked on multiple core backend services, including the CommonService for reusable components and error handling, the EurekaService for service discovery using Netflix Eureka, and the GatewayService for API routing and token validation with Spring Cloud Gateway. He designed NotificationService, where he implemented RabbitMQ message queuing and email dispatch functionality. He implemented the circuit breaker architecture to ensure the service is fault tolerant. Responsible for dockerizing the entire application.

Shaline Raghupathy was primarily responsible for the ChatService and MasterService. She implemented real-time messaging using WebSockets and Kafka, with Redis as a caching layer for performance optimization in the ChatService. For the MasterService, she developed the course and student management features, integrating PostgreSQL for data persistence and transaction handling. Created a scalable front-end system with a robust system architecture.

Nithishh Saravanan collaborated with Naveen on the AuthService, focusing on Keycloak integration for role-based authentication and JWT management. His contributions included setting up the Keycloak server, defining roles, and ensuring secure user verification across the services. Worked on building admin, teacher and student frontend dashboard efficiently handling CRUD operation.

The team collectively ensured smooth integration of all services, using GitLab for version control and collaboration. Regular team meetings were conducted to track progress and address development challenges.

Reflections

The development of **Ctrl+Edu** involved multiple phases, from initial system design to implementation and testing. Throughout the project, the team encountered several challenges and gained valuable insights while working with distributed technologies and microservices.

Key Challenges Faced:

The most significant challenges arose during the integration of distributed components. Synchronizing multiple microservices such as the AuthService, ChatService, and MasterService required careful handling of data flow, service communication, and authentication. Real-time chat implementation with **Kafka** and **WebSocket** presented difficulties in message persistence and guaranteed delivery, especially when handling multiple concurrent users. Additionally, configuring **Keycloak** for role-based access control and ensuring proper token propagation across services proved complex during the early stages.

Solutions Implemented:

To address service communication challenges, the team implemented **Netflix Eureka** for service discovery and **Spring Cloud Gateway** for centralized request handling, ensuring dynamic service registration and load balancing. The real-time messaging issues were resolved by introducing **Redis** as a temporary caching layer for message retention and **Kafka** for event-driven streaming. For security, **Keycloak** was fine-tuned with proper realm and client configurations, ensuring secure JWT handling and consistent role management across the system.

What Could Be Done Differently:

If given the opportunity to restart the project, the team would focus more on **early-stage service orchestration** and **container management** using tools like **Kubernetes** to streamline microservice deployments and scaling. More thorough planning in API design and standardizing data models across services would further reduce integration challenges. Additionally, more proactive **load testing** and stress simulations during development would have helped in better understanding system behavior under heavy traffic.

Lessons Learned:

Working on **Ctrl+Edu** deepened the team's understanding of distributed systems and microservice orchestration. The project highlighted the importance of **asynchronous messaging** and **event-driven architectures** in handling real-time communication effectively. Technologies like **Kafka**, **RabbitMQ**, and **Redis** demonstrated their value in achieving fault tolerance and scalability. The experience with **Keycloak** also reinforced the importance of centralized identity management in distributed systems. Finally, the use of **GitLab** for collaboration and version control proved invaluable for team-based development in a distributed environment.