Publish article application

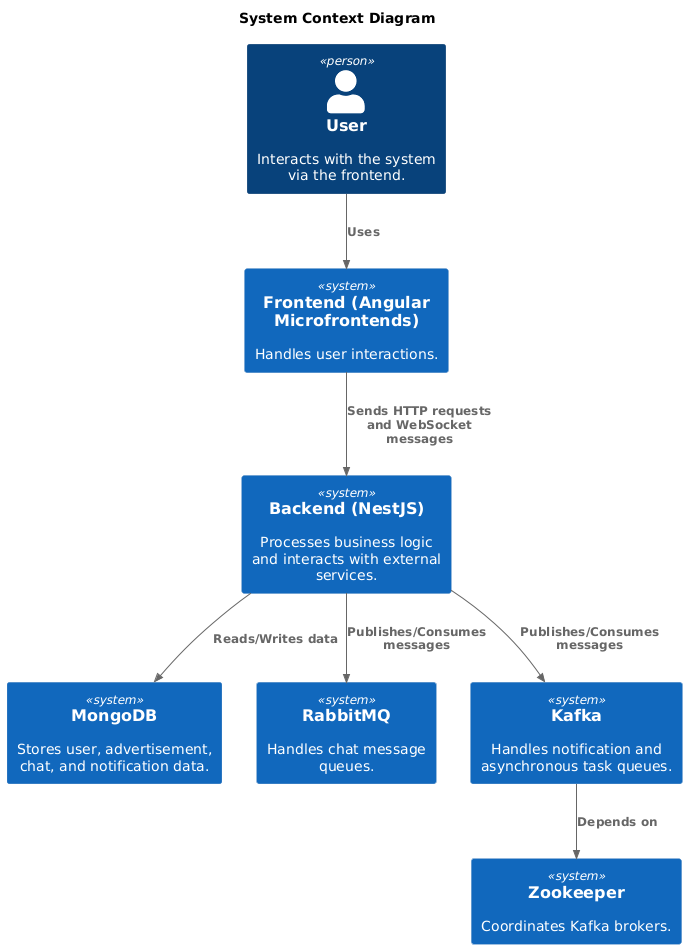
Ionescu Anda

Gr. 258

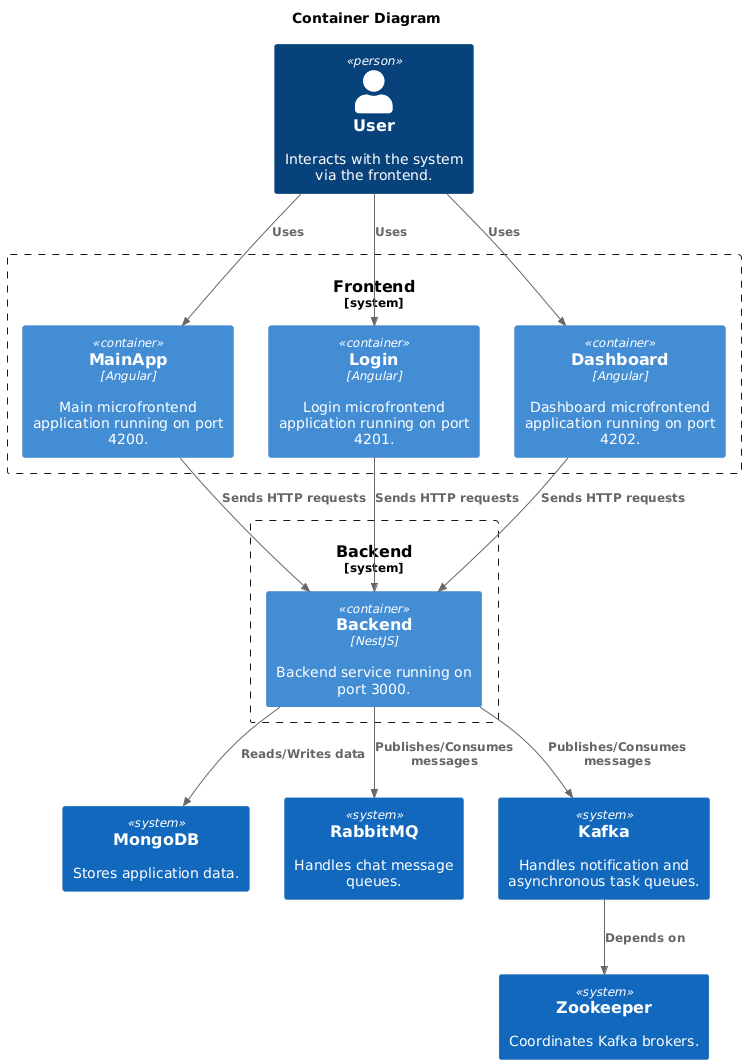
**Description:**

The application is a microservices-based platform designed to manage users, advertisements, real-time chat, and notifications. It is built using NestJS for the backend and Angular for the frontend, with a modular architecture that ensures scalability and maintainability.

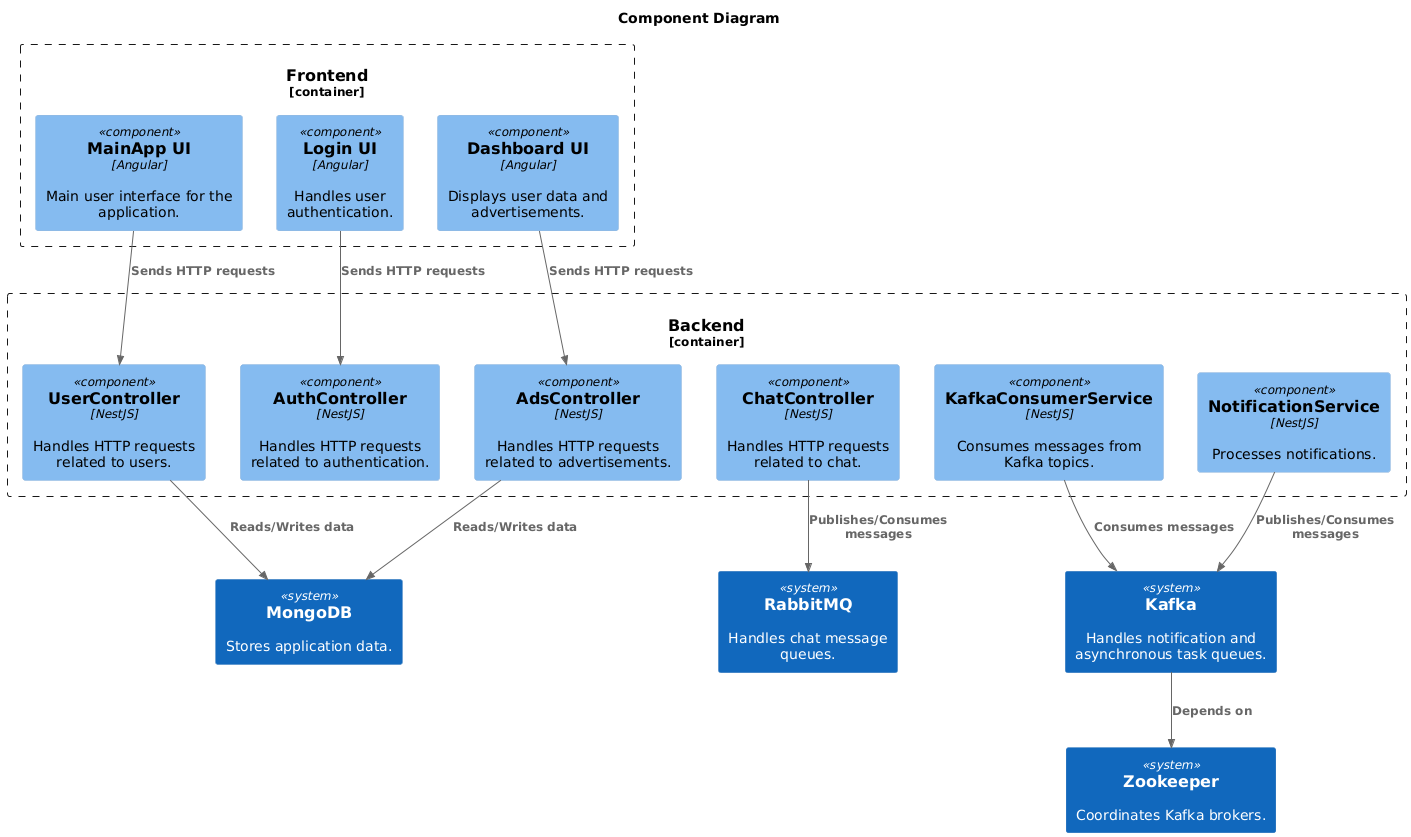
**Level 1 - System context diagram:**



**Level 2 - Container diagram:**

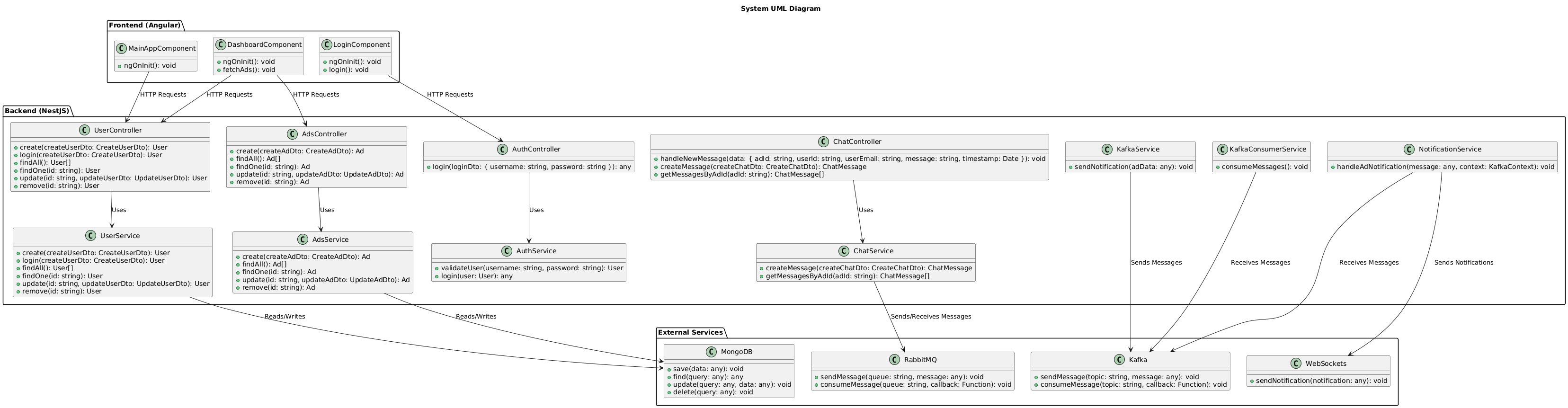


**Level 3 - Component diagram:**



**Level 4 - Code:**

**User microservice UML diagram:**



**Technologies used in the application:**

For communication between the frontend and backend, the application uses HTTP requests and WebSockets for real-time interactions. The backend is built using NestJS, a Node.js framework, which provides a modular and scalable architecture for microservices. For real-time chat functionality, the application uses RabbitMQ as a message broker to handle chat messages and WebSockets to enable real-time communication between users. For notifications and asynchronous task processing, the application uses Kafka as an event streaming platform, which allows for efficient message publishing and consumption. To ensure coordination between Kafka brokers, the application relies on Zookeeper. For data storage, the application uses MongoDB, a NoSQL database, to store user data, advertisements, chat messages, and notifications. For deployment, the application uses Docker to containerize each service (frontend, backend, MongoDB, RabbitMQ, Kafka, and Zookeeper) and Docker Compose to orchestrate the containers.

**SOA Patterns Used**:

The application leverages Service-Oriented Architecture (SOA) patterns to ensure modularity, scalability, and maintainability. The Microservices Pattern divides the application into independent services, each handling a specific domain such as user management, advertisements, chat, and notifications, promoting loose coupling and scalability. The Integration Pattern seamlessly incorporates external services like RabbitMQ for message queuing and Kafka for event streaming, enabling real-time communication and asynchronous processing. The Facade Pattern simplifies complex interactions with third-party services; for example, the KafkaConsumerService acts as a facade for consuming Kafka messages, and the ChatGateway handles WebSocket connections for real-time chat. Additionally, the application employs an Event-Driven Architecture, using events to trigger actions across services. Chat messages are published to RabbitMQ, and notifications are streamed via Kafka, ensuring real-time responsiveness. Together, these patterns create a robust, scalable, and maintainable system that supports real-time features and seamless integration with external services.